

# 4D COMPILER

## 4<sup>th</sup> DIMENSION

By David Hemmo

451

## 4D COMPILER

Documentation by Aline Hemmo  
and David Brandt

Design by Patrick Chédal C&C

Copyright© 1990 ACI and ACIUS, Inc.

### SOFTWARE LICENSE AGREEMENT

ACI grants you a non-transferable, non-exclusive license to use this copy of the program and accompanying materials according to the following terms:

#### LICENSE:

##### You may:

- a) use the program on only one computer at a time;
- b) make one (1) copy of the program in machine readable form solely for backup purposes, provided that you reproduce all proprietary notices on the copy;
- c) physically transfer the program from one computer to another, provided that the program is used on only one computer at a time; and
- d) transfer the program onto a hard disk only for use as described above provided that you can immediately prove ownership of the original diskettes.

##### You may not:

- a) use the program in a network unless you pay for a separate license for each terminal or workstation from which the program will be accessed;
- b) modify, translate, reverse engineer, decompile, disassemble, create derivative works based on, or copy (except for the backup copy) the program or accompanying materials;
- c) rent, transfer or grant any rights in the program in any form or accompanying materials to any person without the prior written consent of ACI which, if given, is subject to the conferee's consent to the terms and conditions of this license; or
- d) remove any proprietary notices, labels or marks on the program and accompanying materials.

This license is not a sale. Title and copyrights to the program, accompanying materials and any copy made by you remain with ACI

#### TERMINATION

Unauthorized copying of the program (alone or merged with other software) or the accompanying materials, or failure to comply with the above restrictions will result in automatic termination of this license and will make available to ACI other legal remedies. Upon termination you will destroy or return to ACI the program, accompanying materials and any copies.

#### LIMITED WARRANTY AND DISCLAIMER

THE PROGRAM AND ACCOMPANYING MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

ACI does not warrant that the functions contained in the program will meet your requirements or that the operation will be uninterrupted or error free. The entire risk as to the use, quality, and performance of the program is with you. Should the program prove defective, you, and not ACI, assume the entire cost of any necessary repair.

However, ACI warrants the diskettes on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt. The duration of any implied warranties on the diskettes is limited to the period stated above. ACI's entire liability and your exclusive remedy as to the diskettes (which is subject to you returning the diskettes to ACI or an authorized dealer with a copy of your receipt) will be the replacement of the diskettes or, if ACI or the dealer is unable to deliver a replacement diskette, the refund of the purchase price and termination of this Agreement.

SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

#### LIMITATION OF LIABILITY

IN NO EVENT WILL ACI BE LIABLE FOR ANY DAMAGES, INCLUDING LOSS OF DATA, LOST PROFITS, COST OF COVER OR OTHER SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES ARISING FROM THE USE OF THE PROGRAM OR ACCOMPANYING MATERIALS, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY. THIS LIMITATION WILL APPLY EVEN IF ACI OR AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOU ACKNOWLEDGE THAT THE LICENSE FEE REFLECTS THIS ALLOCATION OF RISK. SOME STATES DO NOT ALLOW LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

#### GENERAL

This Agreement will be governed by the laws of France. In any dispute arising out of this Agreement, ACI and you each consent to the jurisdiction of both the state and federal courts of France.

Use, duplication or disclosure by the U.S. Government is subject to restrictions stated in paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

Licensor: **ACI, 5 Rue Beaujon, 75008 Paris, France**

This Agreement is the entire agreement between us and supersedes any other communications with respect to the program and accompanying materials.

If any provision of this Agreement is held to be unenforceable, the remainder of this agreement shall continue in full force and effect.

If you have any questions, please contact: **ACI Customer Service, (33) 1 42 27 37 25** or write us at the above address.

**SIGN AND MAIL THE REGISTRATION CARD TODAY.**

Return of the registration card is required to receive any product updates and notices of new versions or enhancements.

All trade names referenced are the trademark or registered trademark of their respective holder.

4D Compiler, 4th DIMENSION, and the 4th DIMENSION impossible 4 logo are trademarks of ACI and ACIUS, Inc.

## CONTENTS

<b>Chapter 1</b>	<b>Introduction.....</b>	<b>1</b>
	Using 4D Compiler.....	4
	How this Manual is Organized.....	5
	How to Use this Manual.....	5
	Interpreted Mode versus Compiled Mode.....	6
	Interpreted Mode.....	6
	Compiled Mode.....	6
	4D Compiler and 4th DIMENSION.....	7
	Interactive Debugging.....	7
	Why Compile your Database?.....	8
	Speed of Execution.....	8
	Checking your Code.....	9
	Protecting your Applications.....	10
	A Practical Exercise.....	10
	Installing 4D Compiler.....	10
	Running the Simulation.....	10
	Compiling the Database.....	11
	Using a Compiled Database.....	13
	Your Turn.....	14
<b>Chapter 2</b>	<b>User's Guide to the Compiler.....</b>	<b>15</b>
	Using 4D Compiler with 4th DIMENSION.....	17
	4D Compiler Menus.....	18
	New.....	18
	Open.....	19
	Recompile.....	21
	Close.....	21
	Save.....	21
	Save As.....	21
	Revert to Saved.....	21
	Quit.....	21
	Compilation Options.....	22
	Compiled Database Name.....	22
	Merge with Runtime.....	23
	Error File.....	24

Symbol Table .....	25
Range Checking .....	25
Script Manager .....	26
Advanced Warning .....	26
Processor Type .....	27
General Comments about Options .....	27
Starting a Compilation .....	28
Compilation Process .....	28
Copying the Database .....	29
Typing Variables .....	29
Compilation .....	31
Using the Compiled Database .....	33
<b>Chapter 3 Diagnostic Aids .....</b>	<b>35</b>
Symbol Table .....	37
List of Global Variables .....	37
List of Local Variables .....	39
List of Procedures .....	39
Error File .....	40
Types of Messages .....	40
Using the Error File .....	43
Range Checking .....	45
How and When to Use Range Checking .....	46
Diagnosing Anomalies .....	47
<b>Chapter 4 Preparing a Database for Compilation .....</b>	<b>49</b>
<b>Data Types of Variables and Arrays .....</b>	<b>51</b>
The Symbol Table .....	52
Typing Variables with 4D Compiler .....	52
Compiler Directives .....	54
When to Use Compiler Directives .....	54
Optimizing Code .....	56
Using Compiler Directives with the Interpreter .....	57
Where to Place your Compiler Directives .....	58
The C_STRING Compiler Directive .....	59
Summary .....	60



<b>Chapter 5</b>	<b>Typing Guide.....</b>	<b>61</b>
	Global Variables.....	63
	Conflict Between Two Uses .....	63
	Conflict Between Use and a Compiler Directive .....	64
	Conflict Stemming from Implicit Retyping .....	64
	Conflict Between Two Compiler Directives .....	65
	Local Variables.....	66
	Conflicts in Array Variables.....	66
	Changing Data Types of Array Elements.....	67
	Changing the Number of Dimensions of an Array .....	67
	String Arrays.....	68
	Implicit Retyping .....	68
	Layout Variables .....	68
	Layout Variables Typed as Numeric .....	68
	Graph Variable .....	69
	External Object Variable .....	69
	Layout Variables Typed as Text .....	69
	Pointers .....	70
	External Procedures .....	71
	Handling Parameter Passing .....	72
	Using Pointers to Avoid Retyping.....	72
	Parameter Indirection .....	74
	Reserved Variables .....	75
	System Variables .....	75
	Quick Report Variables .....	76
<b>Chapter 6</b>	<b>Details about Certain Commands.....</b>	<b>77</b>
	Arrays .....	79
	COPY ARRAY .....	79
	SELECTION TO ARRAY and ARRAY TO SELECTION .....	80
	LIST TO ARRAY and ARRAY TO LIST .....	80
	Using Pointers in Array-Related Commands.....	81
	Communications .....	81
	Data Entry .....	82
	Exceptions.....	82
	Macintosh Desktop.....	83
	Math.....	83

On a List .....	84
Strings .....	84
Structure Access .....	84
Variables and Miscellaneous .....	85
Undefined .....	85
SAVE VARIABLE and LOAD VARIABLE .....	85
CLEAR VARIABLE .....	86
Get Pointer .....	87
EXECUTE .....	88
TRACE and NO TRACE .....	89
Pointers with the Following Commands .....	89
<b>Chapter 7 Optimization Hints .....</b>	<b>91</b>
Using Compiler Directives to Optimize Code .....	93
Numeric Variables .....	93
Strings .....	94
Various Observations .....	95
Pointers .....	96
Using Comments .....	96
<b>Appendix A 4D Compiler Messages .....</b>	<b>97</b>
Warnings .....	99
Advanced Warnings .....	100
Error Messages .....	101
Typing .....	101
Syntax .....	103
Parameters .....	105
Operators .....	106
External Procedures .....	107
General Errors .....	107
Range Checking Messages .....	109
Compiler Messages .....	110
<b>Appendix B Customizing Application Icons .....</b>	<b>111</b>
<b>Index .....</b>	<b>115</b>



# **1 INTRODUCTION**

## INTRODUCTION

---

4D Compiler is the compiler for 4th DIMENSION. It provides numerous features and options that are not found in conventional compilers. The 4D Compiler:

- Systematically analyzes your database and provides extensive error diagnostic warnings and messages
- Provides dynamic checking of the database while the compiled database is running
- Supports interactive debugging with 4th DIMENSION
- Compiles your procedures and scripts and generates true assembly code
- Can generate code optimized for the 68020/30 and 68881/82
- Can merge a compiled database with a copy of 4D Runtime, creating a stand-alone (double-clickable) application

The database to be compiled is opened through the standard open-file dialog box. From then on, the compiler takes care of everything: It duplicates the structure file, systematically analyzes the database, generates descriptive and diagnostic reports, translates the copy's procedures into machine language, and, possibly, generates an error file. You use the new structure file exactly the same way you use the original (uncompiled) structure file, except that you cannot enter the Design environment.

Operating speed of a compiled database is dramatically increased, by a factor ranging from 3 to 1000, and even beyond for certain operations.

4D Compiler generates true assembly code, adapted to the machine's microprocessor, 68000 or 68020/68030, while taking into account their mathematical coprocessor (68881 or 68882).

*NOTE: 4D Compiler compiles any database developed in Version 2. However, to use a compiled database you must have at least Version 2.1 of 4th DIMENSION. Contact ACIUS or ACI about obtaining an update.*



4D Compiler is the compiler for 4th DIMENSION. It provides numerous features and options that are not found in conventional compilers. The 4D Compiler:

- Automatically analyzes your database and provides extensive error diagnostic warnings and messages.
- Provides dynamic checking of the database while the compiled database is running.
- Supports interactive debugging with 4th DIMENSION.
- Compiles your procedures and scripts and generates true assembly code.
- Can generate code optimized for 4th DIMENSION.

## Using 4D Compiler

For optimal use of 4D Compiler, you should keep in mind the following points:

- Poorly-written procedures that do not perform well running in interpreted mode may not do much better when compiled. Furthermore, poorly-written code may not be compilable: 4D Compiler detects the errors that prevent compilation.
- A database that works well under 4th DIMENSION may not always be compilable. The 4th DIMENSION interpreter is tolerant of certain kinds of semantic and syntactic inconsistencies. 4D Compiler is exceptionally adaptable — as compilers rarely are — but it cannot be as flexible as an interpreter. This is due to the essential differences between interpreted and compiled modes.

Unlike the 4th DIMENSION interpreter, the compiler requires that each variable be assigned exactly one data type. That is, the compiler must be able to positively type each variable. No ambiguity with regard to type is permitted. If a variable is used as a Real in one statement and as a Text variable in another statement, the compiler will generate an error message.

You cannot change the type of a global variable anywhere in the database and the type of a local variable within the procedure or script in which it appears.

Much of the work in preparing a database for compilation centers around meeting these conditions. You will find that the compiler contains several excellent diagnostic tools that aid in locating and fixing data typing problems.

Sometimes you have latitude in how you type certain variables. Specifically, there are three types of numeric variables — real, integer, and long integer — and two types of alphanumeric variables — text and string. You can improve the performance of your compiled database by choosing the optimal type for each variable. Using integer variables can make procedures execute faster than using reals. Similarly, using fixed-length strings rather than text variables can make your code execute faster.

## How this Manual is Organized

This manual includes the following chapters:

- **Chapter 1: Introduction.** This chapter introduces you to the compiler and explains the differences between compiled and interpreted programs.
- **Chapter 2: User's Guide.** This chapter describes the features and options available in 4D Compiler.
- **Chapter 3: Diagnostic Aids.** This chapter describes three compiler tools that are useful when debugging a database: the Symbol Table, the Error File, and Range Checking.
- **Chapter 4: Preparing a Database for Compilation.** This chapter describes the main principles you need to follow when writing a database that is to be compiled.
- **Chapter 5: Typing Guide.** This chapter identifies the most frequent sources of typing conflicts.
- **Chapter 6: Details about Certain Commands.** This chapter provides additional information about certain commands in the language that are relevant only to compilation.
- **Chapter 7: Optimization Hints.** This chapter contains some hints that can be used to optimize the performance of your code.

Two appendices contain supplemental information about the compiler. Appendix A contains complete lists of messages displayed by the compiler, with example code that produces each message. Appendix B contains information on customizing your application's icon.

## How to Use this Manual

Read Chapters 1, 2, and 3 to learn how the compiler works. You may want to try to compile some of your databases as you explore the features and options described in Chapter 2. It is unlikely that your databases will be compilable on the first attempt; study the error messages and the symbol table generated by the compiler as you read the next group of chapters. Refer to Appendix A for explanations of error messages generated by the compiler.

Read Chapters 4, 5, and 6 for detailed information on how to write code that is compilable. Finally, read Chapter 7 for hints on how to optimize compilable code.

## Interpreted Mode versus Compiled Mode

This section describes the fundamental differences between interpreted and compiled programs.

The computer is a device in which commands are written using only "0"s and "1"s. The heart of the machine, the microprocessor, is incapable of understanding any other language. This binary language is called *machine language*.

A program written in any high-level computer language (C, Pascal, BASIC, 4th DIMENSION, and so forth) is first translated into machine language, so as to be understandable to the computer's microprocessor.

There are two ways to do this:

- The statements can be translated during execution; the program is said to be *interpreted*
- The statements are translated as a whole, before program execution, in which case the program is said to be *compiled*

### Interpreted Mode

When a series of statements is executed using an interpreter, the process can be broken down as follows:

- The program reads a statement in the program's own language
- It translates the statement into machine language
- It executes the statement

This cycle is executed for each of the statements in the program. Whenever you use 4th DIMENSION, the statements in your procedures and scripts are interpreted.

### Compiled Mode

An application that uses a compiler works differently. A compiled program is translated in its entirety prior to execution. This process results in a new file that contains a set of statements in machine language. This set is saved for repeated use. That is, the translation is performed only once and the compiled version of the program is available for repeated execution.

A disadvantage of this approach is that any debugging or modification of the program must be done on the original or *source* code. Then the

entire program must be recompiled. The compiled version of the program is not directly accessible to the developer. For this reason 4D Compiler leaves your original database intact, so that you can make changes to it and recompile.

The advantages to compiled mode are considerable, and are described in the section "Why Compile your Database?", later in this chapter.

## 4D Compiler and 4th DIMENSION

4D Compiler compiles all global, file, and layout procedures, and all scripts. It generates a new structure file that is used in exactly the same manner as the original structure file. You can manage data in either the User or Runtime environments. The 4D Compiler retains your original structure file and creates a new (compiled) file for your use.

When 4D Compiler finds errors or ambiguities, it usually cannot compile the database. It reports the errors in a text file. You correct the errors in the original (uncompiled) database and then recompile.

## Interactive Debugging

A unique feature of 4D Compiler is that the error file can be used to debug a database interactively. Both the error file and the uncompiled database can be opened simultaneously in 4th DIMENSION. A new 4th DIMENSION menu command, Next Compiler Error, automatically finds and highlights the next compiler error and displays the error message or warning.



## Why Compile your Database?

The first benefit of compilation is, of course, speed of execution. There are two further benefits directly linked to compilation:

- Systematic code check
- Database protection

## Speed of Execution

The increased speed is due to two characteristics of compiled code:

- Direct code translation, once and for all
- Direct access to variable and procedure addresses

## Direct and Final Code Translation

4D Compiler translates your code and saves it as an executable file. The time required in interpreted mode to translate all the statements is saved whenever you use the database.

Here is a simple case that illustrates this point. Take the case of a loop containing a sequence of statements that is repeated 50 times:

```
For (i;1;50)
  `Sequence of statements
End for
```

In an interpreted database each statement in the sequence is translated 50 times. Using the compiler, the translation phase for each statement is eliminated. For every statement in the sequence, we save 50 translations.

Here is another example. Suppose your database includes a Startup procedure. This procedure is executed each time you enter your database.

After compilation, you save the time required for translating the Startup procedure.

These are only two examples. This enhancement actually affects all the statements in all procedures and scripts.

## Direct Access to Variable and Procedure Addresses

In interpreted procedures, variables are accessed through a name. Therefore, 4th DIMENSION needs to access the name in order to access the variable's value.

In the compiled code, the compiler attaches an address to each variable, writes the variable's address directly in the code, and goes directly to that address.

*NOTE: Operations requiring disk access may not be affected, because their speed of operation is limited by the rate of transmission between computer and hard disk.*

## A Comment about Comments

Comments are not translated, so they do not appear in the compiled code. Thus, comments do not affect the execution time of a compiled database.

## Checking your Code

Before attempting to compile a database, the compiler performs a logical and lexical check of your procedures and scripts. It systematically checks your code and notes possible ambiguities, whereas 4th DIMENSION does this when the procedure is executed. An important difference is that the compiler systematically checks *all* code, regardless of whether it is actually executed in typical use.

Suppose that one of your procedures contains a series of tests, as well as sequences of statements to be executed. It is unlikely that you would fully test for all cases if the number of tests were very large. In this instance, a syntax error in an untested case might not show up until an end-user encounters the case.

When you compile a database, the entire database is scanned and each statement is analyzed. Where there is an abnormality, the compiler detects it and generates an error message or a warning. These messages are written to an error file that can be opened within 4th DIMENSION to facilitate interactive debugging. For complete information about interactive debugging, see the section "Using the Error File Interactively with 4th DIMENSION" in Chapter 3.

## Protecting your Applications

The compiled database is a copy of your original database, except that the compiled database shuts off access to the Design environment. The benefits are:

- The database structure cannot be modified, intentionally or by accident
- Your procedures are protected

## A Practical Exercise

Here is a brief tutorial exercise that demonstrates the increase in speed that the compiler provides.

### Installing 4D Compiler

If you have not already done so, insert the 4D Compiler program diskette in your disk drive. Copy the 4D Compiler icon onto your hard disk. At the same time, transfer the Simulationf folder that comes with it.

4D Compiler works on Macintosh computers that have at least one megabyte of RAM and a hard disk.

*NOTE: The minimum memory requirement for 4D Compiler is 700K. If you need to compile a large database (several megabytes), increase the memory size allotted to the compiler. This will provide better working conditions for compilation.*

Using the Macintosh's cache memory (accessible from the Control Panel) considerably speeds up the compiling process. Allow at least 700K for 4D Compiler, and some additional cache memory (32K to 1 megabyte; anything in excess of 1 megabyte would be wasted).

## Running the Simulation

The "Simulationf" folder on the diskette is a small 4th DIMENSION database. It is meant to provide a brief introduction to the compiler.

### 1. Start 4th DIMENSION and open the Simulation database.

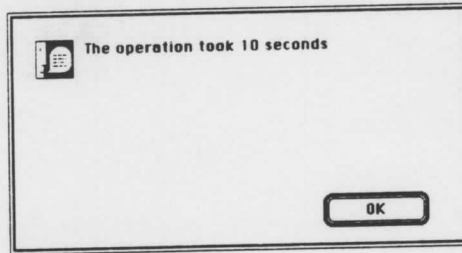
The database opens in the Runtime environment.

### 2. Choose Demonstration from the File menu.

4th DIMENSION displays a list of records. You will now run a procedure that performs operations on its elements.

### 3. Click the Simulation button.

The application executes a procedure that performs operations on reach value in the database. After completion of the operation, an alert displays the time it took.



*NOTE: The time varies depending on the model of Macintosh you are using.*

### 4. Click OK and choose Quit from the File menu.

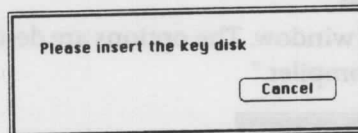
4th DIMENSION quits and returns to the Finder.

## Compiling the Database

In this section, you will compile the database and repeat the performance test.

### 1. Start 4D Compiler by double-clicking the program icon.

Before opening, 4D Compiler asks you to insert your program disk.



*NOTE: If the 4D Compiler key disk is already in the drive when the compiler is started, 4D Compiler will not request the disk.*

### 2. Insert your program disk.

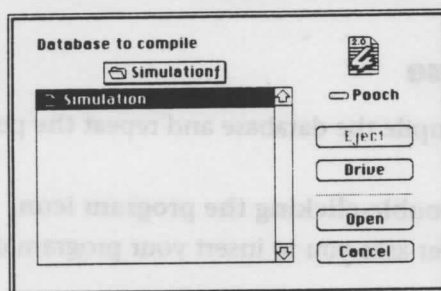
The compiler ejects your master disk automatically and opens.

### 3. Choose New from the File menu.

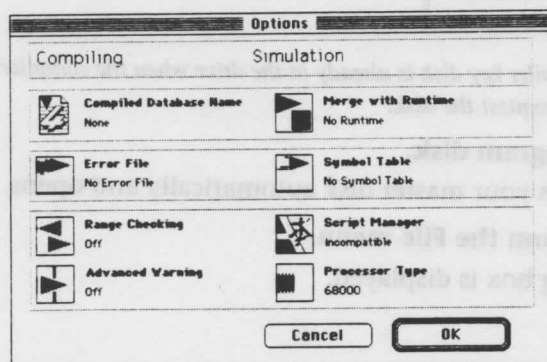
An open-file dialog box is displayed.



4. Open the database structure in the Simulationf folder:



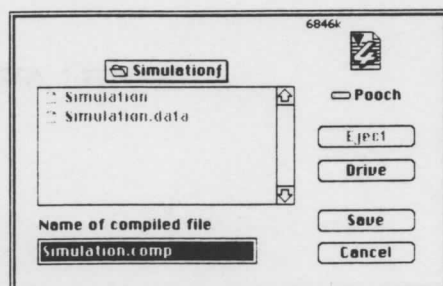
The compiler displays the Options window. The options are described in Chapter 2, "User's Guide to the Compiler."



For this exercise, you don't need to use any of the options.

5. Click the OK button.

A save-file dialog box is displayed.



Before 4D Compiler can compile the database, you need to give the compiled database a name. The default name is *Simulation.comp*.

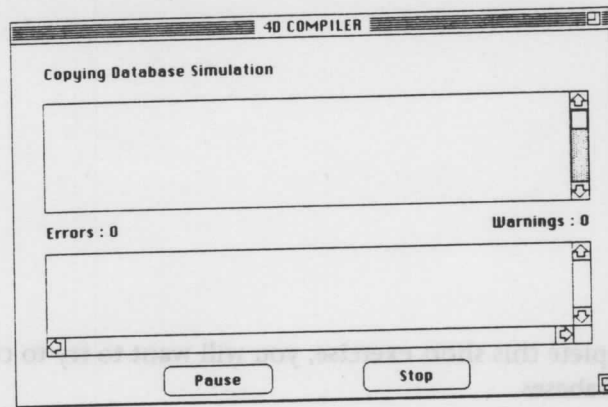
6. Click the Save button.

4D Compiler asks you if you want to save the project. A project is a set of specifications in the Options window. You will find that projects are

very convenient for "real life" work, but for this exercise, you do not need to save a project.

## 7. Click No.

4D Compiler starts the compilation process. During compilation, the 4D COMPILER window is displayed. It shows the progress of the compilation and reports any errors or warnings that it encounters.



The compiler notifies you that it did its job by displaying the message "End of Compilation" at the top of this window.

## 8. Choose Quit from the File menu.

### Using a Compiled Database

To use a compiled database you must have version 2.1 (or later) of 4th DIMENSION.

#### 1. Start 4th DIMENSION, and open the Simulation.comp database.

4th DIMENSION asks you for the data file belonging to the compiled database.

#### 2. Select the Simulation.data file.

Using a compiled database is identical to using an uncompiled one.

#### 3. Go through the steps described in the section "Running the Simulation" and compare performance times.

You will note a dramatic speed increase.

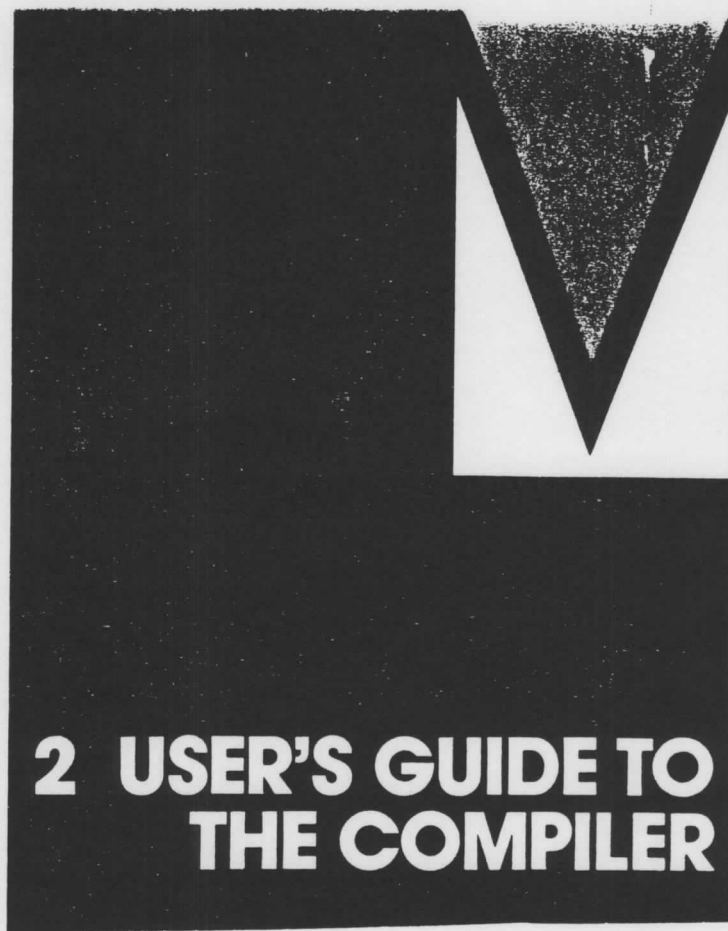
## Your Turn

Once you complete this short exercise, you will want to try to compile one of your databases.

Regrettably, your database may not be compilable on the first attempt. Because the 4th DIMENSION interpreter is more lenient than the compiler, many databases that perform flawlessly in interpreted mode are not compilable on the first try.

The compiler may find dozens of data typing conflicts. However, they are very easy to fix using the compiler's unique interactive debugging.

The following chapter is a user's guide to the compiler. Following that chapter are several chapters that describe the process of debugging and writing code that is optimized for the compiler.



## **2 USER'S GUIDE TO THE COMPILER**



## USER'S GUIDE TO THE COMPILER

This chapter describes how to use 4D Compiler. It describes:

- Using the compiler concurrently with 4th DIMENSION under Multi-Finder
- 4D Compiler menus
- Features and options that are available in the Options window
- The compilation process

4D Compiler options are organized around the concept of a *project*. A project is a group of selected options that you can save to disk. A project can be thought of as a style sheet for the compiler.

Although you do not need to create a project to compile a database, a project makes the process of compiling, debugging, and recompiling quicker and easier.

Using a project, you can:

- Save the name of the compiled structure file
- Create and save the error file and symbol table
- Specify various options, including Merge with Runtime, Range Checking, Advanced Warnings, compatibility with the Script Manager, and target microprocessor.

Each time you recompile a database using a project, 4D Compiler automatically generates the files you specify and uses the options you have invoked. For complete information about the options that are specified using a project, see "Compiler Options," later in this chapter.

### Using 4D Compiler with 4th DIMENSION

A database intended for compilation can be opened by both 4th DIMENSION and the compiler. Using MultiFinder, you avoid having to quit 4th DIMENSION and reopen the database each time you want to compile it.

To compile a database opened under 4th DIMENSION you only have to exit the Design environment by switching to the User or Runtime environments, and then switch to the compiler.

During compilation you cannot use your database. Once compilation is completed, you can return to your database by switching to 4th DIMENSION.

The compiler can work in the background under MultiFinder. You can use another application while the compiler does its job. Even when you are in the Options window you can change applications, and the number of applications you can open simultaneously is limited only by the amount of RAM in your Macintosh.

## 4D Compiler Menus

4D Compiler's menu bar has three menus:

- Apple
- File
- Edit

The Apple menu includes the command "About 4D Compiler." This command displays information about the compiler.

The Edit menu is required for all Macintosh applications. It serves no purpose for 4D Compiler and is provided for use with desk accessories.

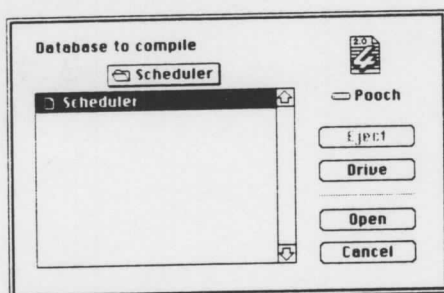
Thus, the only menu that contains commands relating to the compiler is the File menu. It contains eight items.

File	
New	⌘N
Open	⌘O
Recompile	⌘R
Close	
Save	⌘S
Save as...	
Revert to saved	
Quit	⌘Q

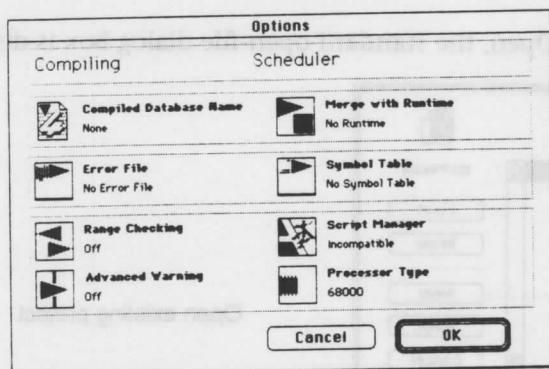
### New

Use this command to create a new project. The project specifies the database to be compiled, as well as all the compiling options you select.

When you choose this command, the compiler displays the standard open-file dialog box shown below. Use this dialog box to open the database to be compiled.



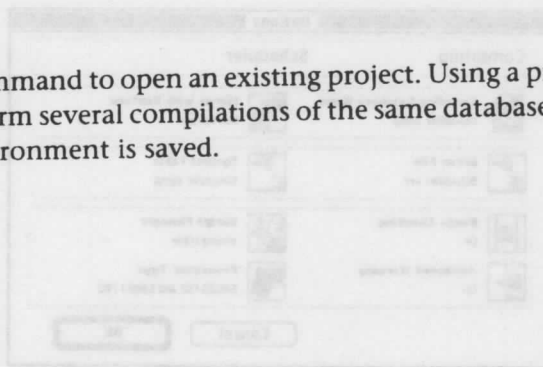
When you create a project, 4D Compiler displays the Options window.



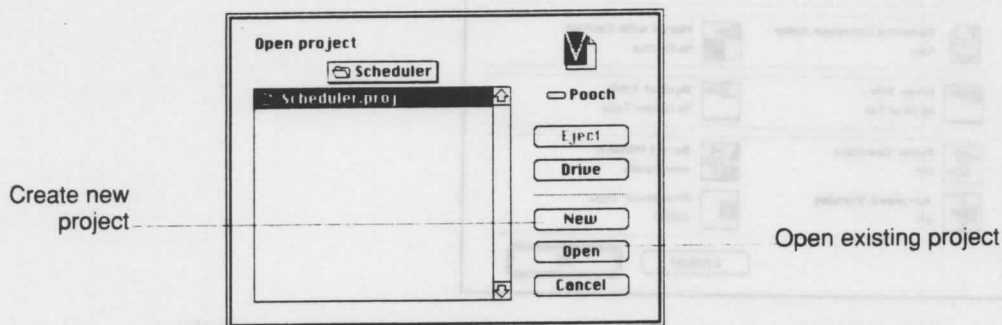
Each of the eight icons in the Options window refers to a compiler option. These options are described in detail in the section, "Compilation Options," later in this chapter.

## Open

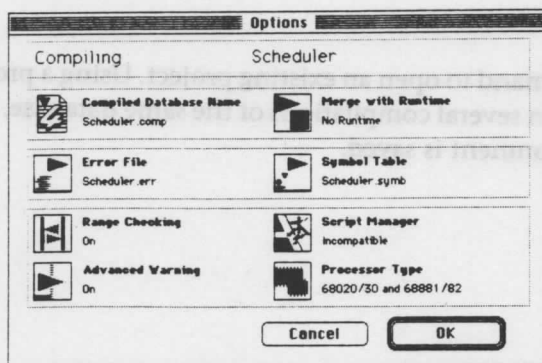
Use the Open command to open an existing project. Using a project, you can quickly perform several compilations of the same database, since the compilation environment is saved.



When you choose Open, the standard open-file dialog box is displayed.



When you select the project you want to open, 4D Compiler displays the Options window for that project. All of the settings for that project are displayed.



If you choose Open from the File menu when you have no existing project, the New button allows you to open a database for compilation.

## Recompiling a Database using a Project

When you open an existing project, 4D Compiler uses all the options defined by the project. Some of these options create files that contain information about the results of the compilation (symbol table, error file, and so forth). Recompiling a project automatically recreates all the related files. If you want to keep files created by a prior compilation, be sure to save them elsewhere before recompiling.

For more information about these files, see the section, "Compilation Options," later in this chapter.

## Recompile

Use the Recompile command to recompile a database that you are in the process of debugging. The Recompile command is intended for use during interactive debugging under MultiFinder. After you have fixed errors in 4th DIMENSION, you can switch to 4D Compiler and recompile your database by choosing Recompile from the File menu. For complete information on interactive debugging, see the section, "Using the Error File interactively with 4th DIMENSION" in Chapter 3.

## Close

Use the Close command to close the current project. If you haven't saved the current project and choose Close, the compiler displays a save-file dialog box, giving you a chance to save it.

## Save

Use the Save command to save the current project, replacing the previously saved version.

## Save As

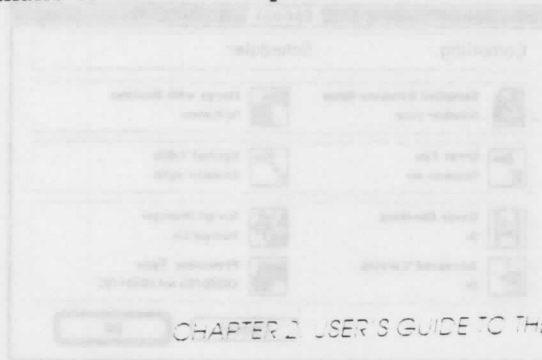
Use the Save As command to save a new project. This command displays the save-file dialog box that allows you to specify a new name for the current project.

## Revert to Saved

Use the Revert to Saved command to return to the last saved version of the current project.

## Quit

Use the Quit command to exit 4D Compiler.





## Compilation Options

The Options window includes eight icons. Each of these icons represents an option that can be selected or deselected. Click an icon to choose one of the actions it controls. A second click cancels the action of the first. All eight icons can be toggled.

Each of the options is described in this section.

### Compiled Database Name

None

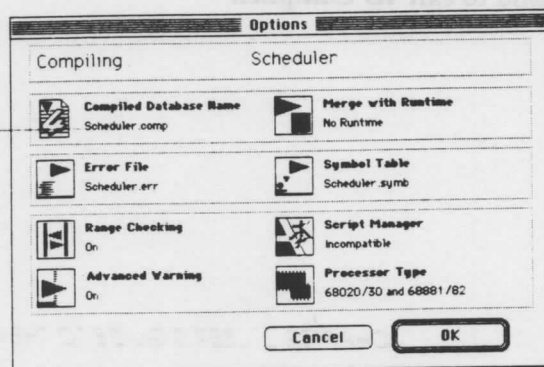


Selected

Use this option to name the compiled structure file. When you click this icon, the create-file dialog box is displayed. The default name for the compiled database is *DatabaseName.comp*.

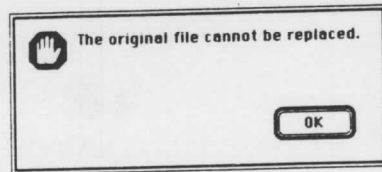
You can change this name as you wish. Click the Save button to save the name of the compiled structure file. After you name the compiled database, it is shown to the right of the icon in the Options window.

Compiled  
database name



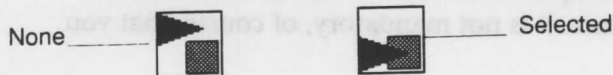
When the original database and the compiled database are located in the same folder, they must have different names. If you attempt to give the compiled database the name of the original one, the compiler displays the standard replace-file dialog box. You do not want to do this, because you need the original file for debugging. Even if you click Yes,

4D Compiler displays an error message and prevents you from overwriting the uncompiled database.



A compiled database is a replica of the original database; it operates exactly like the latter, except that you cannot access the Design environment. Because you need access to the Design environment, 4D Compiler does not let you replace the original file.

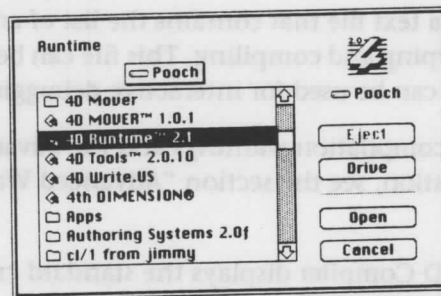
## Merge with Runtime



Use this option to merge your compiled database with a 4D Runtime and create a stand-alone (double-clickable) application. Before choosing this option, you must copy a 4D Runtime onto your hard disk.

***You must purchase a copy of 4D Runtime for each double-clickable application that you create or distribute.***

When you click this option, 4D Compiler displays an open-file dialog box.





Select the Runtime and click Open. When you return to the Options window, your choice of a merge with a Runtime is indicated by the name of the Runtime you selected.

**NOTE:** You must use version 2.1 (or above) of 4D Runtime.

During compilation, the compiler consolidates the Runtime with the database's compiled structure. It is not mandatory, of course, that you select this option.

By default, the double-clickable application is given the generic application icon. You can customize this icon by following the directions in Appendix B.

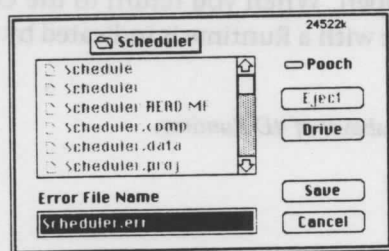
### Error File

None   Selected

Use this option to generate a text file that contains the list of errors and warnings detected during typing and compiling. This file can be opened within 4th DIMENSION and can be used for interactive debugging.

The error file also includes compilation warnings and the advanced warnings. For more information, see the section "Advanced Warning," later in this chapter.

When you click the icon, 4D Compiler displays the standard create-file dialog box. This file's default name is *DatabaseName.err*.



Use the default name if you want to use interactive debugging. Click the Save button. The name of the error file is shown in the Options window.

This file can be used in two ways:

- As a text file, enabling you to keep written records of the messages produced by the compiler
- Within 4th DIMENSION, enabling you to debug your database in real time

For complete information about this file, see Chapter 3 and the list of error messages in Appendix A.

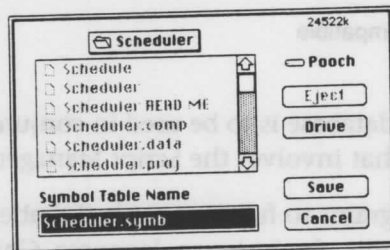
## Symbol Table



Use this option to generate a text file that contains the symbol table of your database. This file contains information concerning all the objects in the database to be compiled. This includes information on global variables, their data types, and the name of the procedure from which the data type was determined.

The symbol table also includes a list of your procedures and functions, the data types of their parameters, and, for functions, the data type of the value returned.

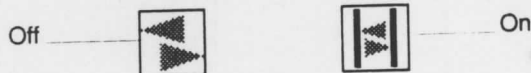
When you click the icon, 4D Compiler displays the standard save-file dialog box. The file's default name is *DatabaseName.symb*.



You can change this name if you wish. Click the Save button. The name of the symbol table is shown in the Options window.

For complete information about the symbol table, see Chapter 3.

## Range Checking



Range Checking provides you with a particularly powerful set of diagnostics. It monitors the execution of procedures while the compiled database is running.

Range Checking does not display messages during compilation. The effects of this option are manifest only while you use the compiled database. It is provided because there are certain types of problems that can only be detected while the database executes. For complete information about Range Checking, see Chapter 3.

## Script Manager

Incompatible



Compatible

Use this option whenever your database is to be used in conjunction with a 4th DIMENSION version that involves the Script Manager.

The Script Manager enables programs to function with alphabets and characters other than the latin ones, for instance: Japanese, Chinese, Arabic, or Hebrew. If your database will be used with a version of 4th DIMENSION that works under Script Manager, select this option.

Otherwise, you should not select Script Manager because your database would be slightly slower.

## Advanced Warning

Off



On

Use this option to generate more extensive diagnostic messages in the error file. There are two cases where it is especially valuable:

- Your code is compilable, and you want your code to be of the highest quality possible
- You are using statements about which the compiler cannot have an opinion. The compiler has made a sensible deduction considering what you have written, but you want to check whether its deductions match your intention.

When this option is selected, the compiler generates informative messages, not error messages. Such messages are automatically written in the error file.

For complete information about this option, see Chapter 3.



## Processor Type

68000



68020/30 and 68881/82

4D Compiler allows you to choose the microprocessor language you want to compile for. Your choice depends on the model of Macintosh on which the compiled database will run rather than the one on which it will be compiled.

Some models of Macintosh use the 68000, such as the Macintosh Plus and SE, and others, such as the Macintosh II family, use the 68020/30 and a mathematical coprocessor (68881/82).

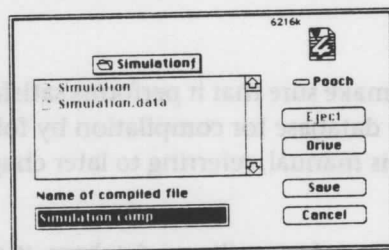
The 68020 and 68030 microprocessors are inherently more powerful than the 68000. This compiling option lets you take advantage of this feature. Moreover, 4D Compiler generates code that takes advantage of the math coprocessors that are installed with the faster microprocessors.

A database compiled for a 68000 will run on computers equipped with a 68000 and also on machines with the 68020 or 68030. However, a database compiled for a 68020/68030 will not run on a 68000-based computer. An error message would be displayed when you start the compiled database.

If you are compiling for a Macintosh that is equipped with a third-party accelerator card that uses the 68020/30 but does not use a mathematical coprocessor, use the 68000 option.

## General Comments about Options

None of the compiling options are mandatory. You can even start a compilation without selecting any option. In this case, when you click the OK button, the compiler asks you to name the compiled database. It displays the save-file dialog box.



You can also create a project without specifying the name of the compiled database. When you start a compilation, 4D Compiler asks you to name the compiled database.

## Starting a Compilation

Once your compiling options are selected, you start the compilation process by clicking the OK button in the Options window. From then on, the compiler works on its own, without your intervention.

*NOTE: If the database is password-protected, 4D Compiler prompts you for the Designer's password when you click OK.*

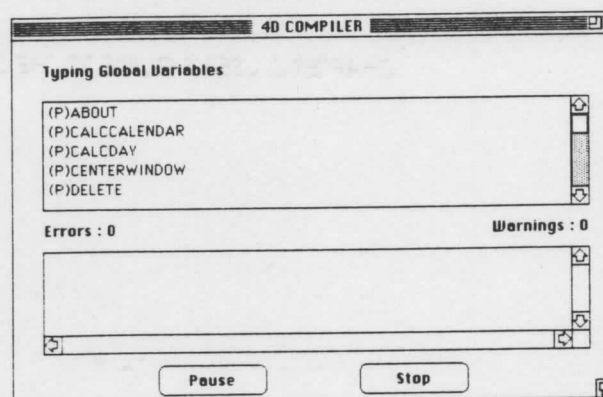
It would stop only if it couldn't find your external procedures. This point is described in the section "Compilation Process," later in this chapter.

If you click the Cancel button in the Options window, 4D Compiler closes the current project, and asks you whether you want to save it, if you have not done so already.

## Compilation Process

Before compiling your database, make sure that it performs satisfactorily in interpreted mode. Prepare the database for compilation by following the guidelines in Chapter 4 of this manual, referring to later chapters in this manual, if necessary.

When the compiler is in the process of compiling a database, it displays the 4D COMPILER window. It looks like this.



In the upper part of the window, the names of procedures and scripts scroll as they are processed by the compiler.

In the lower part of the window, 4D Compiler displays a list of messages and, possibly, errors. Above this window are two counters: the error counter and the warning counter. The counters are automatically incremented throughout the compilation process.

The Pause and Abort buttons enable you to interrupt the process, temporarily (Pause) or permanently (Abort). For more information, see the section "Interrupting the Compilation Process," later in this chapter.

The main phases in the compilation process are:

- Copying the database
- Typing the variables
- Compiling

A small, moving icon on the left side of the upper window informs you of the status of the compilation.

## Copying the Database

Only the database structure file (the file without suffix) is duplicated and compiled. This file contains the procedures and scripts of your database. The data file can be opened by either the compiled or uncompiled structure file.

If you selected the Merge with Runtime option, the compiler also duplicates the Runtime in this phase.


## Typing Variables

In this phase, the compiler creates the symbol table for the compiled database. This phase consists of three passes:

- **The directive typing pass.** This pass identifies the compiler directives and the array declaration statements and uses them to assign data types to variables and arrays.
- **The global typing pass.** This pass types global variables that were not typed in the directive typing pass.
- **The local typing pass.** This pass types the local variables.


Each of the passes is indicated by an icon. If your procedures are small and/or if your computer is very fast, you will barely see them.

### **Directive Typing Pass**

This pass is symbolized by the following icon: 


In this pass, the compiler locates and stores the variables' data types which are declared using compiler directives. In the same way it locates the arrays in the database and stores their data types.

### **Global Typing Pass**

This pass is symbolized by the following icon: 

In this pass, the compiler locates and stores the data types of the global variables that are not declared using compiler directives. The compiler types them according to their use.

### **Local Typing Pass**

This pass is symbolized by the following icon: 

In this pass, the compiler locates and stores the data types of the local variables in the database. It uses compiler directives linked to the local variables, if they exist. Local variables that have no compiler directive are typed according to their use.

### **General Considerations**

This is the phase in which the typing conflicts described in the Typing Guide (Chapter 5) are found.

In cases of conflict, 4D Compiler uses the first occurrence of the variable, within each pass. Global procedures are analyzed in the order in which they are listed in the 4th DIMENSION procedure editor.


Next to be analyzed are file procedures, layout procedures, and scripts attached to layouts, in the order of the files and layouts.

If you change that order, either by sorting or by copying, the messages indicating type conflicts may be different, although the database, when executed, is the same.

## Conclusion of the Typing Phase

Once typing is successfully completed, or if detected errors do not make it impossible to compile the database, the compiler automatically starts the compilation pass, as described in the next section.

If general typing errors were found, 4D Compiler does not proceed with the compilation. Instead, it performs an additional typing pass.

This pass is symbolized by the following icon: 


These errors stem from two sources:

- The compiler encountered two objects with the same name: a procedure and a variable; a variable and an external procedure; or two global procedures. When 4D Compiler finds two variables of different types with the same name, it records the data type conflict, but it does not stop at that level.
- The compiler encountered a global variable that it cannot type.

It is up to you to make corrections with the help of the messages provided by the compiler.

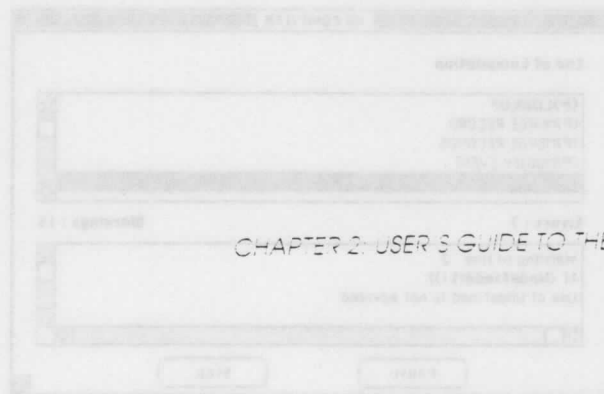
## Compilation

This is the pass in which 4D Compiler translates and saves your procedures in machine language. Here, 4D Compiler continues its search for errors. These are no longer typing errors, but compilation errors (syntax, various inconsistencies, and so forth).

This pass is symbolized by the icon: 

If no errors were found in the compilation pass, the compiled database is created.

If the compiler detects typing or compilation errors, then the compiled database cannot be created and you must first make the corrections.





### Interrupting the Compilation Process

You can interrupt the compiler during typing and compilation by clicking Pause or Stop at any time.

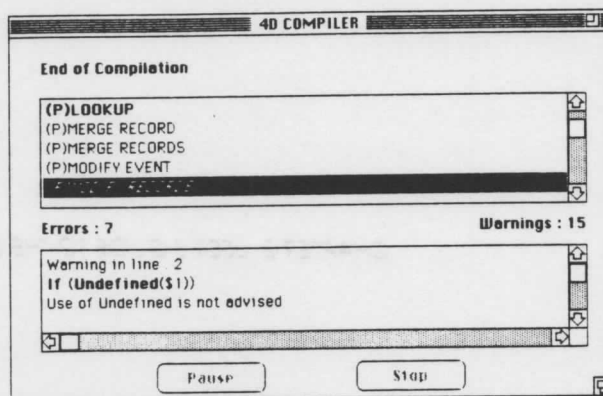
Interruptions are useful whenever the compiler reports errors or warning messages. You can navigate among procedures when messages are attached to them. Navigation is performed in two ways:

- Use the mouse to scroll the list and click the procedure you want
- Use the Tab key to move from one procedure containing errors or warnings to the next. Use Shift-Tab to move in the reverse direction.

The compiler indicates errors and warnings in two ways, and uses them according to the nature of the message:

- When it detects an error in a procedure or script, the compiler sets its name in bold
- When it issues a warning about a statement in a procedure or script, the compiler sets its name in italics
- If a procedure contains both errors and warnings, its name is set in bold.

You can select the procedure by clicking.




The error or warning is described in the lower panel. The three lines of the description are:

- Line number in the procedure
- The line containing the error or warning
- An explanation of the error or warning

## Using the Compiled Database

You open the compiled database in the same way you open any 4th DIMENSION database. However, you cannot enter the Design environment.

If you requested an integration of your compiled database with a 4D Runtime, double-click the application.

The default icon for the application looks like this: . If you want to customize the icon, refer to Appendix B.

Your compiled database executes just like your uncompiled one.

*NOTE: An uncompiled structure file and its compiled version can open the same data file. Specifically, you can use the two structure files on the same network and both can open the same data file.*



## **3 DIAGNOSTIC AIDS**

## DIAGNOSTIC AIDS

The compiler produces three types of diagnostic aids that help make debugging easier:

- **Symbol Table.** Analysis of the database is facilitated by the symbol table. Use it to find your way through your variables quickly. It is a valuable tool for interpreting the error messages reported by 4D Compiler.
- **Error File.** Debugging a database is facilitated by the error file that you can use as a text file or within 4th DIMENSION.
- **Range Checking.** Range checking is a sophisticated tool for monitoring and controlling the execution of your procedures within your compiled applications.

This chapter describes these aids and contains complete information about how to use them.

### Symbol Table

The symbol table is a text file that can be opened with any text or word processor. Information is presented in columns, with tabs separating each column. The symbol table contains complete information about the objects in your database. The document is divided into three parts:

- List of global variables
- List of local variables, in their procedure or script
- List of global procedures and functions, with their parameters, if applicable

### List of Global Variables

Information about each global variable is presented in four columns:

- The first column contains a complete list of global variables and arrays used in your database. These variables are listed in alphabetical order.
- The second column contains the data type of each object. Types were established through compiler directive or determined from the use of the object. If the data type could not be determined, the column is empty.
- The third column lists the number of dimensions, if the object is an array.

- The fourth column contains a reference to the context in which the compiler established the object's data type. If the variable is used in several contexts, the context mentioned is the one used by the compiler to determine its data type.

Here are the symbols used to identify where the variable name was found:

- If the variable was found in a global procedure, the procedure is identified by the name you gave it in 4th DIMENSION, and preceded by (P).
- If the variable was found in a file procedure, the filename is given, preceded by (FP).
- If the variable was found in a layout procedure, the layout name is given, preceded by (LP).
- If the variable was found in a script, the script's name is given, preceded by layout name, filename, and by (S).
- If the variable is an object in a layout and does not appear in any procedure, formula or script, the name of the layout in which it appears is given, preceded by (L).

Here is a list of global variables from a typical database.

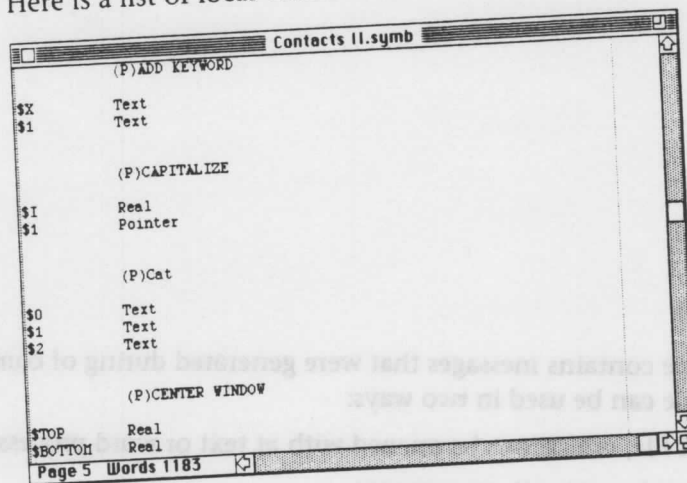
Contacts II.symb			
AADDR	Text	1 dimension	(P)LOOKUP
AADDR2	Text	1 dimension	(P)LOOKUP
ACITY	Text	1 dimension	(P)LOOKUP
ACODE	Long integer	1 dimension	(P)LOOKUP
ADDRESS	Text	1 dimension	(LP)[Contacts] LETTER
ANAME	Text	1 dimension	(P)LOOKUP
ASTATE	Text	1 dimension	(P)LOOKUP
B1	Real		(L)[Letters] SELECT LETTER
B2	Real		(L)[Letters] SELECT LETTER
B3	Real		(L)[Letters] SELECT LETTER
B4	Real		(L)[Letters] SELECT LETTER
BACCEPT	Real		(L)[Contacts] Input
BADD	Real		(L)[Contacts] Contacts OUT 9
BADDEV	Real		(L)[Contacts] Contacts IN 9
BADDRBOOK	Real		(L)[Contacts] Contacts OUT 9
BCANCEL	Real		(L)[Contacts] Input
BDELETE	Real		(L)[Contacts] Contacts IN 9
BDELEKW	Real		(L)[Contacts] APPLY KEYWORD
BDONE	Real		(L)[Contacts] ABOUT
BFIN	Real		(L)[Contacts] APPLY KEYWORD
BFIRST	Real		(L)[Contacts] Input
BFIRSTPAGE	Real		(L)[Contacts] Input
BLABELS	Real		(L)[Contacts] Contacts OUT 9



## List of Local Variables

The list of local variables is sorted by global procedure, file and layout procedure, and script, in the same order as in 4th DIMENSION. Only those procedures that use local variables are shown.

Here is a list of local variables from a typical database.

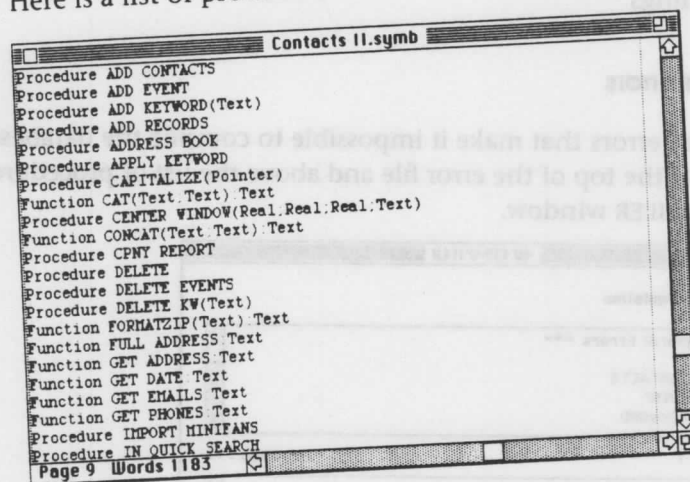


## List of Procedures

A complete list of your global procedures is given at the end of the file, with the data types of their parameters and, for functions, the returned result. This information is presented in the following format:

*Procedure name (parameter data type): result data type*

Here is a list of procedures.



## Error File

The error file contains messages that were generated during of compilation. The file can be used in two ways:

- As a text file, which can be opened with at text or word processor
- Interactively, with 4th DIMENSION

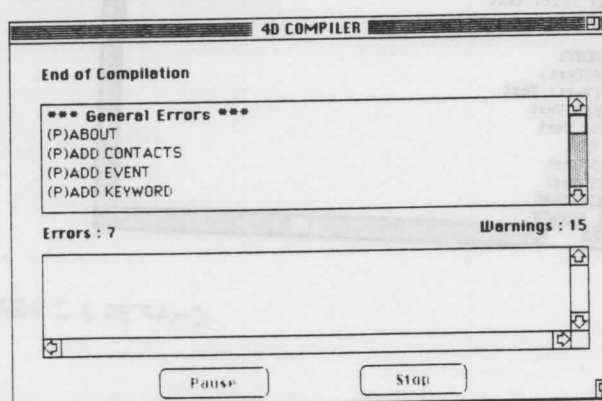
## Types of Messages

4D Compiler generates three types of messages:

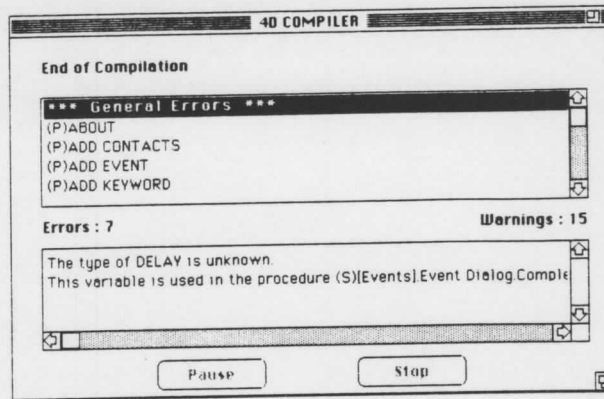
- General errors
- Errors linked to a specific line
- Warnings

### General Errors

These are errors that make it impossible to compile the database. They appear at the top of the error file and above the list of procedures in the 4D COMPILER window.



When you click General Errors, the related messages are displayed in the lower panel, as shown below.



There are two cases in which the compiler reports a general error message:

- The data type of a global variable could not be determined
- Two different kinds of objects have the same name

In the first instance, 4D Compiler could not perform a particular typing anywhere in the database, and, in the second instance, 4D Compiler was unable to decide whether to associate a given name with one object rather than with another.

These errors are called general errors because they cannot be linked to any specific procedure or script. Appendix A contains a list of general errors.

### Errors Linked to a Specific Line

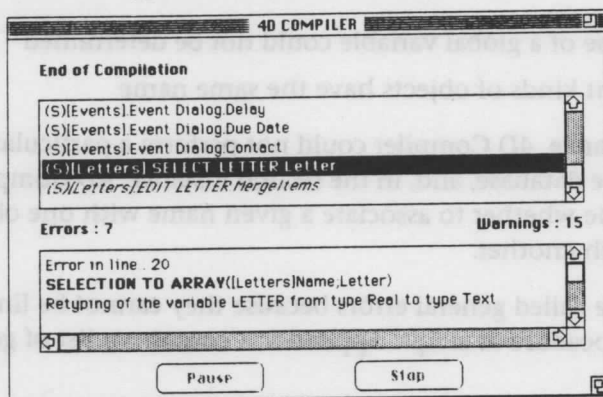
These errors are displayed in context — the line in which they were found — with an explanation.

The compiler reports this type of error when it encounters an expression in which it sees an inconsistency, either data type or syntax related.

Procedures or scripts that contain such errors appear in bold, in the upper window. When you select one of those procedures, information about all errors in the procedure are displayed in the lower panel, in the following manner:

- Line number
- Statement containing the error, as it appears in your procedure
- Description of the error

Here is a typical error in a line of code.

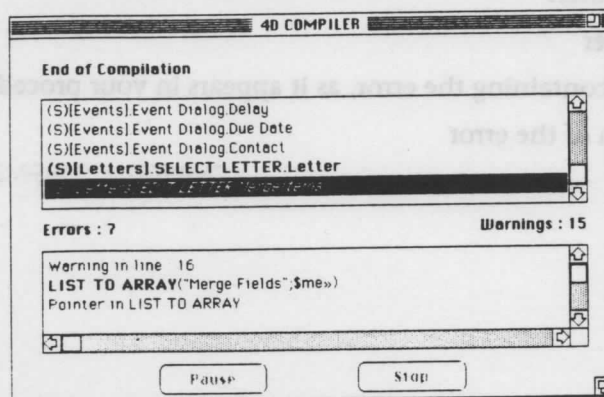


Appendix A of this manual contains the list of these errors.

## Warnings

Warnings are not errors. Warnings do not prevent the database from being compiled. They point out areas to which the compiler wishes to draw your attention because they have a potential for error.

Procedures or scripts in which warnings have been detected appear in italics in the upper window. When you select one of those procedures, the list of warnings are displayed in the lower panel.



The information is presented in the following order:

- Line number
- Statement to which the warning refers, as it appears in your procedure
- Explanation of the warning

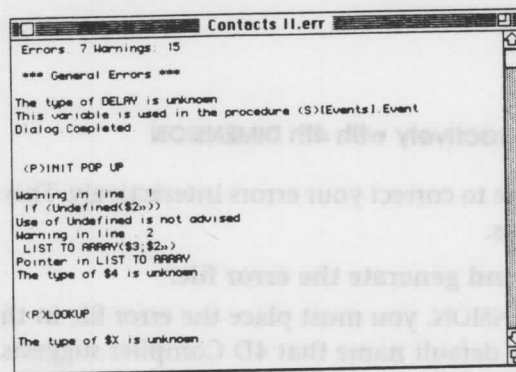
If you had chosen the Advanced Warning option, other warnings would also be displayed. Appendix A contains a complete list of warnings.

## Using the Error File

The error file can be used in two ways, as a text document, and within 4th DIMENSION.

### Using the Error File as Text

When you open an error file using a text editor, it looks like this.



The structure of the error file is as follows:

- The number of errors and warnings is at the top of the file
- The general errors are listed next
- All other errors and warnings are listed last, sorted by procedure or script, in the same order as in 4th DIMENSION

These errors and warnings are listed using the following format:

- The line number in the procedure
- The line containing the error or warning
- A diagnostic that describes the error



## Using the Error File Interactively with 4th DIMENSION

You can use the error file to correct your errors interactively. This section describes how to do this.

### 1. Run the compiler and generate the error file.

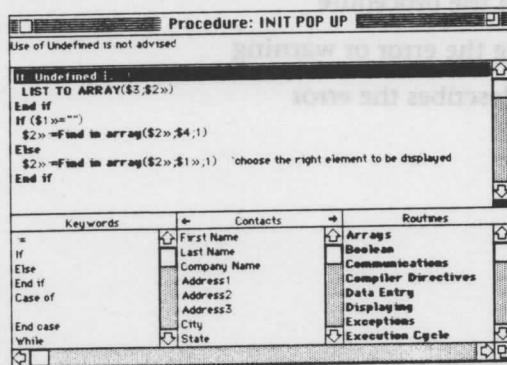
To open it in 4th DIMENSION, you must place the error file in the database folder and use the default name that 4D Compiler suggests.

### 2. Start 4th DIMENSION and open your uncompiled database in the Design environment.

Two commands are automatically added to the Use menu for interactive debugging. The two commands are: Next Compiler Error and Stop Browsing Error File.

### 3. Choose Next Compiler Error from the Use menu.

4th DIMENSION automatically opens the first procedure or script in which 4D Compiler found an error or warning.



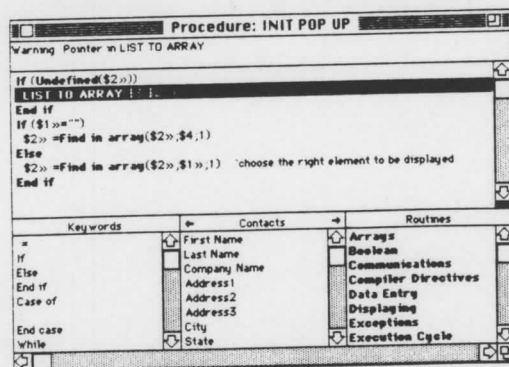
At the top of the window the nature of the error or warning is described. In the procedure, the affected line is highlighted.

### 4. Debug the error that the compiler spots.

If the message is a warning, you may not need to modify your code.

5. Move to the next error or warning by choosing Next Compiler Error from the Use menu.

The next error or warning is displayed.



In this manner, you can correct all errors quickly, without having to spend time locating each error manually.

Use the Stop Browsing Error File command to end the debugging session. When you leave the Design environment, the error file is automatically closed.

## Range Checking

While all the other options operate during the compilation process, Range Checking begins its work only when you run the compiled database. That is, you receive this set of messages while your compiled database is running.

Range checking is an "in situ" controller; it evaluates the status of objects in the database at a given time. Here is an example of how Range Checking works.

Suppose that you declared the array MyArray as Text. The number of elements in MyArray may vary depending on the current procedure.

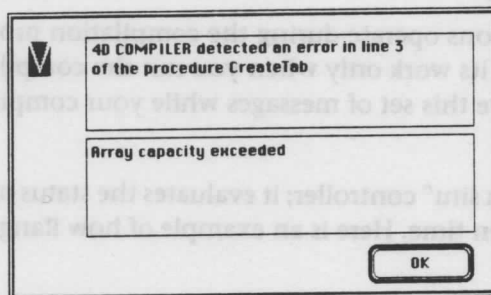
Suppose you want to assign the value "Hello" to element 5 of MyArray. Therefore you write:

```
MyArray{5}:="Hello"
```

If at that time MyArray has 5 elements or more, everything is fine. Assignment proceeds normally. However, if MyArray has less than 5 elements at that moment, your assignment no longer makes sense.

A situation like this cannot be detected at compilation, because of the presupposition that the procedures are executing. The compiler would not know the circumstance in which this procedure is called. Only Range Checking enables you to monitor what is actually happening while your database is in use.

In the example above, the compiler would display a message like this from within 4th DIMENSION.



It is easy to see why Range Checking is especially valuable when arrays, pointers, and strings of characters are being processed.

The messages displayed by the compiler when you request Range Checking are listed in Appendix A of this manual.

### How and When to Use Range Checking

When you request Range Checking, your compiled database becomes substantially slower than it is without this option. Therefore, it should not be requested for the "final" compilation of a finished database.

It should be used as an additional phase in the process of developing and debugging your databases. It is assumed that you will run a final compilation without Range Checking.

The compiler's most obvious purpose is to speed up databases. An equally important goal is to help you in the enhancement of your application's reliability. We therefore recommend that you consider a development phase that includes Range Checking.

## Diagnosing Anomalies

This section provides some background information on the value of Range Checking.

Suppose you notice anomalies in the running of your databases. Before you start speculating about their possible sources, remember the assistance provided by the compiler.

Potential anomalies are:

- 4th DIMENSION displays its own error messages. If possible, correct this error in your database according to instructions provided by 4th DIMENSION. If the latter are too general, compile your database again with the Range Checking option. Retest your database. At the location where the 4th DIMENSION message was displayed, you will see a more informative message from the compiler.
- Your compiled database does not perform exactly like your uncompiled database. Take a closer look at the Advanced Warning messages. Possibly, you may want to add the Range Checking option.
- Your database performs smoothly in interpreted mode. After compilation, an error creates a system crash or a return to the Finder. You find yourself in a situation typical of Assembler or Pascal programmers. If you are well versed in the use of a Macintosh debugger, this tool can help you find the name of the procedure in which you crashed. If you are not familiar with a debugger, a simpler solution is to recompile with the Range Checking option. Chances are good that your problem will be detected.



## **4 PREPARING A DATABASE FOR COMPILATION**



## PREPARING A DATABASE FOR COMPILE

The 4th DIMENSION interpreter gives you a great deal of freedom in the way you can name variables and procedures. An interpreter can be more tolerant of certain types of conflicts and inconsistencies than a compiler.

When writing a database for the compiler, you need to follow these basic principles:

- Each variable, global procedure, and external procedure must have a unique name
- Each variable must be assigned (implicitly or explicitly) exactly one data type

These principles are, simply, good programming practice. They do not limit the operations that can be performed in compiled 4th DIMENSION databases. They are introduced only because it is essential that the compiler be able to identify, without any ambiguity, each object in the database.

The objects that are most likely to create ambiguities are variables. This chapter, therefore, is devoted to them.

Ambiguities stem from two major sources:

- Using the same name for two different variables
- Assigning (implicitly or explicitly) different data types to the same variable

Before creating the compiled database, the compiler makes extensive checks and, when it finds ambiguities, generates diagnostic messages that make it easy to debug databases that were not written with these guidelines in mind.

### Data Types of Variables and Arrays

All variables have a data type. There are 11 data types for variables: Boolean, String, Date, Integer, Longint, Graph, Time, Picture, Real, Pointer, and Text.

For Array type variables, there are 9 data types: Array Boolean, Array String, Array Date, Array Integer, Array Longint, Array Picture, Array Real, Array Pointer, and Array Text.

## The Symbol Table

In interpreted mode, a variable can have more than one data type. This is possible because the code is interpreted rather than compiled.

4th DIMENSION interprets each statement separately and comprehends its context.

When you work in a compiled environment, the situation is different. While interpretation is performed line by line, the compilation process looks at a database in its entirety. The compiler's approach is the following:

- The compiler systematically analyzes the objects in the database. The objects are global, layout and file procedures, and scripts.
- The compiler scans the objects to determine the data type of each variable used in the database, and it generates the table of variables and procedures (the symbol table)
- Once it has established the data types of all variables, the compiler translates (compiles) the database.

But it cannot compile the database unless it can determine the data type for each of the variables.

If the compiler comes across the same variable name and two different data types, it has no reason to favor any particular one. In other words, to type an object and give it a memory address, the compiler must know the precise identity of that object, i.e., its name and its data type. The compiler determines its size from the data type.

For every compiled database, the compiler creates a map that lists, for each variable, its name (or identifier), its location (or memory address), and the space it occupies (indicated by its data type). This map is called the *symbol table*.

## Typing Variables with 4D Compiler

The most straightforward and unambiguous way to type variables is to use the compiler directive commands in the language. However, it is not necessary to use a compiler directive for every variable.

4D Compiler is clever enough to free you from the drudgery of having to declare every variable in your database — something you need to do with conventional compilers. Whenever possible — as long as there is no ambiguity — 4D Compiler determines the type of a variable from the way it is used.

For example, if you write:

```
V1:=12.5
```

the compiler determines that variable V1 is data type Real.

By the same token, if you write:

```
V2:="This is a sample phrase"
```

the compiler determines that V2 is a Text type variable.

The compiler is also capable of establishing the data type of a variable in less straightforward situations. Here is an example:

```
V1:=12.5
```

```
V2:="This is a sample phrase"
```

```
V3:=V1
```

Here, the compiler concludes that V3 is the same type as V1.

Here is a similar example:

```
V4:=2*V2
```

Here, the compiler concludes that V4 is of the same type as V2.

4D Compiler also determines the data type of your variables according to calls to 4th DIMENSION commands, and according to your procedures. For example, if you pass a Boolean type parameter and a Date type parameter to a procedure, 4D Compiler assigns the Boolean type and the Date type to the local variables \$1 and \$2 in the called procedure.

When the compiler determines the data type by inference, it never assigns the limiting data types: Integer, Longint, or String. The default type assigned by 4D Compiler is always the widest possible. For example, if you write:

```
Number:=4
```

the compiler assigns the Real data type to Number, even though 4 happens to be an integer. In other words, the compiler does not rule out the possibility that, under other circumstances, the variable's value might be 4.5.

If it is appropriate to type a variable as Integer, Longint, or String, you can do so using a compiler directive. It is to your advantage to do so, because these data types occupy less memory and performing operations on these data types is faster.

## Compiler Directives

The 4th DIMENSION Language Reference contains a description of the following compiler directive commands:

C\_STRING (length; variable1 {;...; variableN})  
C\_BOOLEAN (variable1 {;...; variableN})  
C\_DATE (variable1 {;...; variableN})  
C\_INTEGER (variable1 {;...; variableN})  
C\_TIME (variable1 {;...; variableN})  
C\_PICTURE (variable1 {;...; variableN})  
C\_LONGINT (variable1 {;...; variableN})  
C\_REAL (variable1 {;...; variableN})  
C\_POINTER (variable1 {;...; variableN})  
C\_TEXT (variable1 {;...; variableN})

These commands enable you to explicitly declare the variables used in your databases. They are used in the following manner:

### **C\_BOOLEAN** (Var)

Through such directives you inform the compiler to create a variable Var that will be a Boolean.

Whenever an application includes compiler directives, 4D Compiler finds them and, thus, avoids guesswork. A compiler directive has priority over deductions made from assignments or use.

*NOTE: There are ten compiler directives whereas there are eleven types of variables. There is no specific compiler directive for graphs since there can never be any ambiguity about the data type of a graph variable.*

## When to Use Compiler Directives

Compiler directives are useful on two occasions:

- The compiler is unable to determine the data type of a variable from context
- You do not want the compiler to determine a variable's type from use

## Cases of Ambiguity

Sometimes the compiler cannot determine the data type of a variable. Whenever it cannot make a determination, 4D Compiler generates an appropriate error message. There are two major causes that prevent the 4D Compiler from determining the data type:

- Multiple data types
- Unable to determine a type

### Multiple data types

If a variable has been retyped in different statements in the database, the compiler generates an error that is easy to fix.

The compiler selects the first variable it encounters and arbitrarily assigns the data type it had previously assigned to the next occurrence of the variable.

Here is a simple example:

```
Variable:=TRUE           in procedure A
Variable:="The moon is green" in procedure B
```

If procedure A is compiled before procedure B, the compiler considers the statement "Variable:="The moon is green"" as a data-type change in a variable previously encountered. The compiler notifies you that retyping has occurred. It generates an error for you to correct. In most cases, the problem can be fixed by renaming the second occurrence of the variable.

### Unable to Determine a Data Type

This case arises when a variable is used without having been declared, and within a context that doesn't provide information about its data type. Here, only a compiler directive can guide the compiler.

This phenomenon occurs primarily within two contexts: when pointers are used, or with a command with more than one syntax.

- **Pointers.** A pointer cannot be expected to return a data type except its own. Consider the following sequence:

```
Var1:=5.2                (1)
Pointer:=»Var1           (2)
Var2:=Pointer»           (3)
```

Although (2) defines the type of variable pointed to by the pointer Pointer, the type of Var2 is not determined. During the compilation pro-



cess, 4D Compiler is able to recognize a pointer, but it has no way of knowing to what type of variable it is pointing. Therefore, it cannot deduce the data type of Var2. A compiler directive such as

**C\_REAL** (Var2)

is necessary.

■ **Multi-syntax commands.** When you use a multi-syntax command, the compiler cannot guess which syntax and parameters you have selected. Here is an example:

The FIELD ATTRIBUTES command accepts two syntaxes:

**FIELD ATTRIBUTES** (file number; field number; type; length; index)

**FIELD ATTRIBUTES** (field pointer; type; length; index)

You must use compiler directives to type the variables passed to the command if they are not typed according to their use elsewhere in the database.

## Optimizing Code

You can speed up your procedures by using compiler directives to type numeric variables as Integer or Longint or text variables as String. Here is a common example.

Suppose you need to increment a counter using a local variable. If you don't declare the variable, 4D Compiler assumes that it is a Real. If you specify that it is a Longint, the compiled database would perform more efficiently. A Real takes 10 bytes of memory; but if you typed the counter a Longint, it would need only 4 bytes. Incrementing a 10-byte counter takes longer than incrementing a 4-byte one. Also, calculations on Reals are slower than calculations on integers. Calculations on Reals are performed in Apple's SANE environment, and this takes longer than processing calculations on integers.

*NOTE: Introducing more compiler directives than necessary has no negative impact. Compiler directives are not translated when the database is compiled.*

## Using Reals and Strings

If you assign a real value to a variable that you declared an Integer or assign a string of 30 characters to a variable that was declared a 10-character one, 4D Compiler assigns values according to your directives. If you assign a real number to an integer variable, it takes the integer part of the value. Similarly, if you assign a 30-character string to a variable typed as

a 10-character string, the compiler takes the first 10 characters. It does not regard either assignment as a type conflict.

Here is an example. If you write:

```
C_INTEGER (vInteger)
vInteger:=2.5
```

4D Compiler takes the integer part of the number (2 instead of 2.5).

*WARNING: In 4th DIMENSION versions preceding 2.1, the vInteger variable would have taken the value 2.5, and not 2. To avoid any surprises, retest your databases using 2.1, compiled and uncompiled.*

Here is an example that deals with strings. In the following sequence:

```
C_STRING (10;MyString)
MyString:="It is a beautiful day"
```

4D Compiler just takes the first ten characters of the constant, "It is a be".

## Using Compiler Directives with the Interpreter

Compiler directives are not required for databases that are not compiled. The interpreter automatically types each variable according to how it is used in each statement, and a variable can be freely retyped throughout the database.

Because of this flexibility, it is possible that a database can perform differently in interpreted and compiled modes. For example, if you write:

```
C_LONGINT (MyInt)
```

and elsewhere in the database, you write:

```
MyInt:=3.1416
```

In the compiled database, the value 3 would be assigned to MyInt. Normally, the interpreter would assign the value 3.1416. This assignment would cause the same database to give different results in interpreted and compiled modes.

Versions 2.1 (and above) of 4th DIMENSION contains a special feature that reduces the chances of discrepancies such as this. The 4th DIMENSION interpreter uses compiler directives to type variables. When the interpreter encounters a compiler directive, it types the variable according to the directive. If a subsequent statement tries to assign an incorrect value (e.g., assigning an alphanumeric value to a numeric variable) the assignment will not take place. However, no error message will be generated.

In the above example, MyInt is assigned the same value (3) in both the interpreted and compiled modes, *provided the compiler directive is interpreted prior to the assignment statement.*

The order in which the two statements appear is irrelevant to the compiler because it first scans the entire database for compiler directives. The interpreter, however, is not systematic. It interprets statements in the order that they are executed. That order, of course, can change from session to session, depending on what the user does. For this reason, it is important to design your database so that your compiler directives are executed prior to any statements containing variables that are declared.

### Where to Place your Compiler Directives

Placing a compiler directive is a simple operation. Two strategies for global variables are open to you:

- Use the directive in the procedure or script in which it first appears. Be sure to use the directive the very first time you use the variable, in the first procedure to be executed.
- Declare your global variables in the Startup procedure

*NOTE: The 4th DIMENSION Language Reference states that the procedure that contains your compiler directives does not have to be executed. If the procedure is not executed, the compiler will use the directives to type variables (as usual) but you may get different results when using the database in interpreted and compiled modes.*

Local variables cannot be declared using this strategy because each local variable only exists in the procedure in which it appears. For local variables, use the following strategy:

- Declare your local variables in the procedure in which they appear.

Local variables cannot be typed globally. The relevant compiler directives should appear prior to the first use of the local variable in the procedure.

*REMINDER: A global variable in 4th DIMENSION is a variable whose value is accessible throughout the database. Its name always begins with one of the 26 letters of the alphabet. A local variable is a variable that exists only within a procedure. Its name begins with the \$ character.*

## The C\_STRING Compiler Directive

The C\_STRING command uses a different syntax than the other directives because it accepts an additional parameter — the maximum string length.

**C\_STRING** (*length*; *variable1* {;...; *variableN*})

Because C\_STRING concerns fixed-length strings, it is natural to specify the maximum length of such strings. In a compiled database, you must specify the length of the string with a constant, rather than a variable. Here is an example.

In an interpreted database, the sequence

TheLength:=15

**C\_STRING** (TheLength;TheString)

is acceptable.

4th DIMENSION interprets Length, then replaces Length with its value in the C\_STRING compiler directive statement.

However, the compiler uses this command when typing variables with no specific assignment in mind. Thus, it is not in a position to know that TheLength equals 15. Not knowing the string's length, it cannot keep a space for it in the symbol table. Therefore, with compilation in mind, a constant should be used to specify the length of the declared string of characters. For example, a statement such as this should be used:

**C\_STRING** (15;TheString)

The same rule applies to declaring fixed string arrays, which are typed with the command:

**ARRAY STRING** (*length*; *array name*; *size*)

The parameter that indicates string lengths in the array must be a constant.

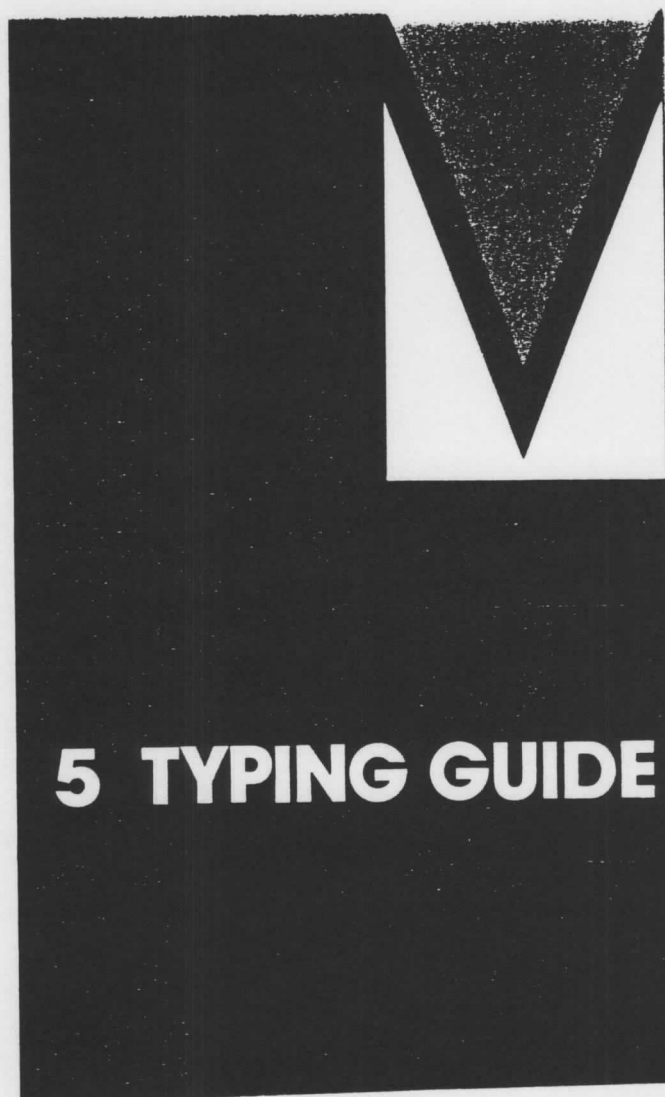
*NOTE: Do not confuse the length of an Alphanumeric field, which has a maximum of 80 characters, with a fixed string variable. The maximum length of a string declared by a C\_STRING directive, or belonging to an ARRAY STRING is between 1 and 255.*

## Summary

This chapter provides an overall description of the compiler's behavior in its analysis of data types of variables. For a more detailed and practical understanding, we suggest that you read the next two chapters. They illustrate the statements in this chapter and provide:

- A guide through potential conflicts of data types, and how to avoid them. This chapter is called "Typing Guide."
- Special notes about compilation for a group of 4th DIMENSION commands.





## **5 TYPING GUIDE**

## TYPING GUIDE

---

This chapter contains further details about typing conflicts that can prevent a database from being compiled. The chapter is organized according to the type of object that generates the conflict. We consider the following types of objects:

- Global variables
- Local variables
- Arrays
- Layout variables
- Pointers
- External Procedures
- Reserved Variables

### Global Variables

Type conflicts regarding global variables can be categorized as follows:

- Conflict between two uses
- Conflict between use and a compiler directive
- Conflict resulting from implicit retyping
- Conflict between two compiler directives

### Conflict Between Two Uses

The simplest data type conflict is one that stems from a single variable name designating two different objects.

Assume you write:

Variable:=5

and in the same database, you write:

Variable:=TRUE

These two statements generate a data type conflict. Often the problem can be solved simply by renaming one of the variables.

## Conflict Between Use and a Compiler Directive

Suppose you write

Variable:=5

and in the same application, you write:

**C\_BOOLEAN** (Variable)

Because the compiler scans the directives first, it will type Variable as Boolean; but when it finds the statement "Variable:=5", it detects a data type conflict.

You can solve the problem by renaming your variable or modifying the compiler directive.

Using variables of different data types in one expression creates inconsistencies. The compiler points out incompatibilities.

Here is a simple example.

Bool:=TRUE	`4D Compiler determines that Bool is data type Boolean
<b>C_INTEGER</b> (Integer)	
Integer:=3	`Assigning value compatible with compiler directive
Var:=Integer+Bool	`Operation using variables whose data types are incompatible

## Conflict Stemming from Implicit Retyping

Some functions return variables of a very precise data type. Assigning the result of one of such variables to a variable already typed differently creates a data type conflict.

In an interpreted application, you can write:

Ident No:= <b>Request</b> ("Identification Number")	`Ident No is data type Text
<b>If</b> (OK=1)	
Ident No:= <b>Num</b> (Ident No)	`Ident No is now data type Real
<b>SEARCH</b> ([Contacts]ID=Ident No)	
<b>End if</b>	

In this example, you create a type conflict in the third line. In some cases, you have to create an intermediate variable that uses a different name. In other cases, such as this one, your procedure can be structured differently.

Ident No:= <b>Num</b> ( <b>Request</b> ("Identification Number"))	`Ident No is data type Real
<b>If</b> (OK=1)	
<b>SEARCH</b> ([Contacts]ID=Ident No)	
<b>End if</b>	

## Conflict Between Two Compiler Directives

Declaring the same variable through two conflicting compiler directives constitutes a retyping. If, in the same database, you write:

```
C_BOOLEAN (Variable)
```

and

```
C_TEXT (Variable)
```

the compiler detects the conflict and reports an error in the error file. Typically, you can solve the problem by renaming one of the variables.

A data type conflict can arise in the use of **C\_STRING** if you modify the maximum string length. Thus, if you write:

```
C_STRING (5;MyString)
```

```
MyString:="Flour"
```

```
C_STRING (7;MyString)
```

```
MyString:="Flowers"
```

the compiler identifies a conflict because it must provide an adequately-sized location when declaring String variables.

The solution is to use only one compiler directive that gives the maximum length, since the compiler accepts a shorter length. You can write:

```
C_STRING (7;MyString)
```

```
MyString:="Flour"
```

```
MyString:="Flowers"
```

If you like, you can use **C\_STRING** (7;String) twice, i.e.,

```
C_STRING (7;MyString)
```

```
MyString:="Flour"
```

```
C_STRING (7;MyString)
```

```
MyString:="Flowers"
```

but it would be redundant.

## Local Variables

Data type conflicts concerning local variables are identical to those in global variables. The only difference is that consistency must be achieved only within a specific procedure or script.

For global variables, conflicts occur at the general level of the database. For local variables, conflicts occur at the level of the procedure or script. For example, you cannot write in the same procedure:

```
$Temp:="Flowers"
```

and then

```
$Temp:=5
```

However, you can write:

```
$Temp:="Flowers"
```

in procedure P1, and

```
$Temp:=5
```

in procedure P2. Local variables are not globally typed.

## Conflicts in Array Variables

Conflicts concerning an array are never size-related. As in uncompiled databases, arrays are managed dynamically. The size of an array can vary throughout procedures and you do not have to declare a maximum size for an array. Therefore, you can size an array to null, add or remove elements, or delete the contents.

You should follow the following guidelines when writing a database intended for compilation:

- Do not change data types of array elements
- Do not change the number of dimensions of an array
- For a string array, do not change character-string length

*REMINDER: Arrays are always global variables. Therefore, any declaration applies to the entire application.*



## Changing Data Types of Array Elements

If you declare an array as Array Integer, it must remain an integer array throughout the database.

If you write:

```
ARRAY INTEGER (MyArray;5)
ARRAY BOOLEAN (MyArray;5)
```

the compiler cannot type MyArray. Simply rename one of the arrays.

## Changing the Number of Dimensions of an Array

In an uncompiled database, you can change the number of dimensions of an array. When the compiler sets up the symbol table, one-dimensional arrays and two-dimensional arrays are managed differently. Consequently, you cannot redeclare a one-dimensional array as two-dimensional, or vice-versa.

Therefore, in the same database you cannot have:

```
ARRAY INTEGER (Array1;10)
ARRAY INTEGER (Array1;10;10)
```

However, you may write the following statements in the same application:

```
ARRAY INTEGER (Array1;10)
ARRAY INTEGER (Array2;10;10)    `Note different array name
```

And you can also write:

```
ARRAY BOOLEAN (Array;5)
ARRAY BOOLEAN (Array;10)      `Resizes the array
```

The number of dimensions in an array cannot be changed in a database. You can, however, change the size of an array. You can resize one array of a two-dimensional array.

*REMINDER: A two-dimensional array is, in fact, a set of several one-dimensional arrays. For more information, see the chapter "Arrays and Pointers" in the 4th DIMENSION Language Reference.*

## String Arrays

String arrays obey the same rules as fixed strings, and for the same reasons.

If you write:

```
ARRAY STRING (5;Array;10)
```

```
ARRAY STRING (10;Array;10)
```

the compiler detects a length conflict. The solution is simple: Declare the maximum string length. 4D Compiler automatically manages shorter length strings.

## Implicit Retyping

While using commands such as COPY ARRAY, LIST TO ARRAY, ARRAY TO LIST, SELECTION TO ARRAY, ARRAY TO SELECTION, you may change, voluntarily or not, the data type of elements, the number of dimensions, or, in a String array, the string length — and you find yourself in one of the three situations previously mentioned.

The compiler generates an error message; the required correction is usually quite obvious. Examples of implicit array retyping are provided in the next chapter in the discussion of the array commands.

## Layout Variables

Variables created in a layout (e.g., buttons, pop-up menus, and so forth) are always global variables.

In an interpreted database, the data type of such variables is not important. However, in compiled applications it may have to be taken into consideration. The rules are, nevertheless, quite clear:

- You can type layout variables using compiler directives, or
- The compiler assigns the appropriate default data types

The conventions used by the compiler in typing layout variables are given in this section.

## Layout Variables Typed as Numeric

The following layout variables are typed as Reals: Check box, Button, Highlight button, Invisible button, Radio button, Radio Picture, Ruler, Dial, and Thermometer.

You may explicitly declare these variables as Integer or Longint. There is an advantage to doing so: a Real requires 10 bytes, an Integer or a Longint only 4.

The only possible data type conflict for one of these variables could arise if the name of a variable were identical to that of another variable located elsewhere in the database. In this case, rename the second variable.

## Graph Variable

A graph area is automatically data type Graph. This variable never creates a data type conflict. The only possible data type conflict for a Graph-type variable could arise if the name of this variable were identical to that of another variable located elsewhere in the application. In this case, rename the second variable.

## External Object Variable

An External Object is always a Longint. There can never be a data type conflict. The only possible data type conflict for an External Object variable could arise if the name of this variable were identical to that of another variable located elsewhere in the application. In this case, rename the second variable.

## Layout Variables Typed as Text

There are four such variables: Non-enterable variable, enterable variable, pop-up menu, and scrollable area.

These variables are divided into two categories:

- **Simple variables** — Enterable and non-enterable variables
- **Display variables** — Pop-up menus and scrollable areas

### Simple Variables

Their default data type is Text. When used in scripts or procedures, they are assigned the data type selected by you. There is no danger of conflict other than a conflict resulting from assigning the same name to another variable.

### Display Variables

Pop-up menus and scrollable areas are only used to display the contents of arrays. There will be no problem if you follow the basic conventions concerning arrays as discussed earlier.

### Pointers

When you use pointers in your database, you take advantage of a powerful and versatile 4th DIMENSION tool. The compiler preserves all the benefits of pointers.

A pointer can point to variables of different data types. Do not create a conflict by assigning different data types to the same variable.

Be careful not to change the data type of a variable to which a pointer refers. Here is an example of this problem.

```
Variable:=5.3  
Pointer:=»Variable  
Pointer»:=6.4  
Pointer»:=FALSE
```

In this case, your dereferenced pointer is a Real variable. By assigning it a Boolean value, you create a data type conflict. If you need to use pointers for different purposes in the same procedure, make sure that your pointers are defined. Here is an example of the correct use of pointers.

```
Variable:=5.3  
Pointer:=»Variable  
Pointer»:=6.4  
Bool:=True  
Pointer:=»Bool  
Pointer»:=False
```

A pointer is always defined in relation to the object it refers to. That is why the compiler cannot detect data type conflicts created by pointers. In case of conflict, you will get no error message while you are in the typing phase or in the compilation phase.

This does not mean that the compiler has no way of detecting conflicts that involve pointers. The compiler is capable of analyzing some uses of pointers with the Range Checking option. This option is discussed in detail in Chapter 3.

## External Procedures

External procedures can be located in four different places:

- In the System file
- In 4th DIMENSION
- In the structure of your database
- In an external procedures file (proc.ext) located in the same folder as your database

In any case, the compiler needs the procedure's definition, i.e., the number of parameters and their data types. There is no danger of typing errors as soon as the compiler actually finds what you have declared in the database.

Often, the simplest solution is to store your external procedures in the structure file of the database to be compiled. When the compiler duplicates the original database, it automatically copies the external procedures. If the externals are in a proc.ext file in the database folder or in the System file, the compiler does not duplicate the file, but it automatically takes into account the declarations of the procedures.

If you place your externals in 4th DIMENSION, 4D Compiler asks you to specify the location of your external procedures during typing. It displays the standard open-file dialog box. Click the document that contains the externals, and click the Open button.

At that time, the compiler analyzes the definition of your procedure. Here too, there is no danger of confusion at the typing level if your calls are consistent with the declaration of the procedure.

You can pass fewer parameters to an external procedure than it is expecting, provided the missing parameters are at the end.



## Handling Parameter Passing

The handling of the local variables,  $\$0... \$n$ , follows all the rules that have already been stated. As with all other local variables, their data type cannot be altered while the procedure executes.

In this section, we examine two instances that could lead to data type conflicts:

- When you actually require retyping. The use of pointers helps avoid data type conflicts.
- When you need to address parameters by indirection.

## Using Pointers to Avoid Retyping

A variable cannot be retyped. However, it is possible to use a pointer to refer to variables of different data types. Here is an example to illustrate this.

Consider a function that returns the memory size of a one-dimensional array. In all cases but two, the result is a Real; for Array Text and Array Picture, the memory size depends on values that cannot be expressed numerically. For more information, see the section "Managing Arrays," in the *4th DIMENSION Language Reference*.

For Text and Picture arrays, the result is returned as a string of characters. This function requires a parameter: a pointer to the array whose memory size we want to know.

There are two methods to carry out this operation:

- Work with local variables without worrying about their data types; in such case, the procedure runs only in interpreted mode.
- Use pointers, and proceed either in interpreted or in compiled mode.

*MemSize* function, only in interpreted mode.

**\$Size:=Size of array (\$1»)**

**\$Type:=Type (\$1»)**

**Case of**

```

: ($Type=14)                `Array Real
    $0:=8+($Size*10)        ` $0 is a Real (Numeric)
: ($Type=15) `Array is Integer
    $0:=8+($Size*2)
: ($Type=16)                `Array is Long integer
    $0:=8+($Size*4)

: ($Type=18)                `Array is Text
    $0:=String (8+($Size*4) +("Sum of text sizes")) ` $0 is text
: ($Type=19) `Array is Picture
    $0:=String (8+($Size*4) +("Sum of picture sizes"))

```

**End case**

In the above procedure, \$0's data type changes according to the value of \$1; therefore, it is not compatible with the compiler. Here is how to write this procedure using pointers.

*MemSize* function in interpreted and compiled modes

**\$Size:=Size of array (\$1»)**

**\$Type:=Type (\$1»)**

**VarNum:=0**

**Case of**

```

: ($Type=14)                `Array Real
    VarNum:=8+($Size*10)    `VarNum is a Real
: ($Type=15)                `Array Integer
    VarNum:=8+($Size*2)    `VarNum is a Real
: ($Type=16) `Array Longint
    VarNum:=8+($Size*4)    ``VarNum is a Real

: ($Type=18)                `Array is Text
    VarText:=String (8+($Size*4) +("Sum of text sizes"))
: ($Type=19) `Array is Picture
    VarText:=String (8+($Size*4) +("Sum of picture sizes"))

```

**End case**

**If (VarNum#0)**

**\$0:=»VarNum**

**Else**

**\$0:=»VarText**

**End if**

Here is the key difference between the two functions:

- In the first case, the function's result is the expected variable
- In the second case, the function's result is a pointer to that variable

You simply dereference your result.

### Parameter Indirection

4D Compiler manages the power and versatility of parameter indirection. In interpreted mode, 4th DIMENSION gives you a free hand with numbers and data types of parameters. You retain this freedom in compiled mode, provided you do not introduce data type conflicts and that you do not use more parameters than you passed in the calling procedure.

To prevent possible conflicts, parameters addressed by indirection must all be of the same data type.

This indirection is best managed if you respect the following convention: If only some of the parameters are addressed by indirection, those parameters should be passed after the others.

Within the procedure, an indirection address is formatted:  $\$ \{i\}$ , where  $i$  is a numeric variable.  $\$ \{i\}$  is called a *generic parameter*.

Here is an example. Consider a function that takes values, adds them up, and returns the sum formatted according to a format that is passed as a parameter. Each time this procedure is called, the number of values to be added may vary. We must pass the values as parameters to the procedure and the format in the form of a character string. The number of values can vary from call to call.

This function is called in the following manner:

```
MySum ("##0.00";125.2;33.5;24)
```

In this case, the calling procedure will get the string "182.70", the sum of the numbers, and formatted as specified. The function's parameters must be passed in the correct order: First the format and then the values.

Here is the function named *MySum*:

```
$Sum:=0  
For ($i;2;Count Parameters)  
    $Sum:=$Sum+${$i}  
End for  
$0:=String ($Sum;$1)
```

This function can now be called in various ways:

```
MySum ("##0.00";125.2;2;33.5;24)
```

or

```
MySum ("000";1;18;4;23;17)
```

As with other local variables, it is not necessary to declare generic parameters by compiler directive. When required (in cases of ambiguity or for optimization), it is done using the following syntax:

**C\_INTEGER (\$4)**

The command above means that all parameters from the fourth (included) on will be addressed by indirection. All will be data type Integer. Data types of \$1, \$2, and \$3 are irrelevant.

*NOTE: In Version 2, 4th DIMENSION replaced variable indirection in Version 1 with new tools: Arrays and pointers. Version 2 accepts the older procedures for compatibility reasons. Since the compiler was developed for Version 2, the performance of the faster and more powerful tools is optimized. If you use Version 1 indirection, change your procedures accordingly.*

## Reserved Variables

Some 4th DIMENSION variables are assigned a data type and an identity by the compiler. Therefore, you can not create a new variable, a procedure, a function or an external procedure using any of these variables. You can test their values, and use them as you do in interpreted mode.

## System Variables

Here is a complete list of 4th DIMENSION system variables with their data types.

System Variable	Type
OK	Longint
Document	String (255)
FldDelimit	Longint
RecDelimit	Longint
Error	Longint
MouseDown	Longint
KeyCode	Longint
Modifiers	Longint

## Quick Report Variables

When you create a calculated column in a report, 4th DIMENSION automatically creates a variable C1 for the first one, C2 for the second one, C3. and so forth. This process is performed in a transparent manner.

If you use these variables in formulas, keep in mind that, like other variables, C1, C2,...Cn cannot be retyped.

System Variable	Type
OK	Longint
Document	String (255)
RelDefint	Longint
RecDefint	Longint
Error	Longint
MonthDown	Longint
KeyCode	Longint
Modifier	Longint





## **6 DETAILS ABOUT CERTAIN COMMANDS**

## DETAILS ABOUT CERTAIN COMMANDS

4D Compiler expects that the usual syntactic rules for 4th DIMENSION commands are followed. In this respect, it does not require any specific modifications for databases that will be compiled.

Some commands that affect a variable's data type may lead to data type conflicts. Because some commands use more than one syntax, it is to your advantage to know which is the most appropriate one to select.

The commands that fall into these categories are discussed in this chapter. They are grouped by topic in alphabetical order.

### Arrays

Five 4th DIMENSION commands are used by the compiler to determine the data type of an array. They are:

**COPY ARRAY** (*from; to*)  
**SELECTION TO ARRAY** (*field; array*)  
**ARRAY TO SELECTION** (*array; field*)  
**LIST TO ARRAY** (*list; array; {linked array}*)  
**ARRAY TO LIST** (*array; list; {linked array}*)

### COPY ARRAY

The **COPY ARRAY** command accepts two array-type parameters. If one of the array parameters is not declared elsewhere, the compiler determines the data type of the undeclared array from the data type of the declared one.

Such deduction is performed in both cases:

- The array typed elsewhere is the first parameter. The compiler assigns the data type of the first array to the second array.
- The declared array is the second parameter. Here, the compiler assigns the data type of the second array to the first array.

Because the compiler is strict about data types, **COPY ARRAY** can be performed only from an array of a certain data type to an array of the same type.

Consequently, whenever you want to copy an array of elements whose data types are similar, that is, Integer, Longint and Real; or Text and

String, where fixed Strings are of various lengths, you have to copy the elements one by one.

Suppose you want to copy elements from an Integer array to a Real array. You can proceed as follows:

```
$Size:=Size of array (ArrInt)
ARRAY REAL (ArrReal;$Size)      `Set same size for Real array and Integer array
For ($i;1;$Size)
    ArrReal{$i}:=ArrInt{$i}      `Copy each of the elements
End for
```

Remember that you cannot change the number of dimensions of an array during the process. If you copy a one-dimensional array into a two-dimensional array, 4D Compiler generates an error message.

## SELECTION TO ARRAY and ARRAY TO SELECTION

The default type for an undeclared array is Text. The undeclared array will be assigned the data type of the field specified in the command.

If you write:

```
SELECTION TO ARRAY ([MyFile]Intfield:MyArray)
```

the data type of MyArray would be Array Integer (assuming that IntField is an integer field).

If the array has been declared, make sure that the field is of the same data type. Although Integer, Longint and Real are similar types, you may not consider them as equivalent.

If an array was not previously declared and you apply one of the commands that includes a String-type field as a parameter, the default data type assigned to the array is Text.

If the array was previously declared as String or Text, these commands will follow your directives. The same is true for Text-type fields: your directives have priority.

## LIST TO ARRAY and ARRAY TO LIST

These commands apply only to two types of arrays: one-dimensional String arrays and one-dimensional Text arrays.

Neither command requires that the array passed as a parameter be declared.

`Receive Variable example  
**SET CHANNEL** (12;"Document1")  
**RECEIVE VARIABLE** (\$Type)

**Case of**

:\$Type=0)  
    **RECEIVE VARIABLE** (\$String) `Processing variable received  
:\$Type=1)  
    **RECEIVE VARIABLE** (\$Real) `Processing variable received  
:\$Type=2)  
    **RECEIVE VARIABLE** (\$Text) `Processing variable received

**End case**

**SET CHANNEL** (11)

*NOTE: Values returned by the Type function are listed in the 4th DIMENSION Language Reference, under function Type.*

## Data Entry

Type (parameter)

Because each variable in a compiled database has only one data type, this function may seem to be of no use. However, it can be useful when you work with pointers. For example, you may need to know the data type of the variable to which a pointer refers; due to pointer flexibility, one cannot always be sure to what object it points.

## Exceptions

**ON EVENT CALL** (event procedure)  
**ON SERIAL PORT CALL** (serial procedure)  
**ABORT**  
**IDLE**

A command has been added to 4th DIMENSION language to manage exceptions: the command **IDLE**. This command should be used whenever you use the command **ON EVENT CALL** or **ON SERIAL PORT CALL**. This command could be defined as an event management directive.

Only the kernel of 4th DIMENSION is able to detect a Macintosh event (mouse click, keyboard activity, and so forth). In most cases, kernel calls are initiated by the compiled code itself, in a way that is transparent to the user.

On the other hand, when 4th DIMENSION is waiting passively for an event — in a waiting loop for instance — it is clear that there will be no call.

Here is an example:

```

`MouseClicked Procedure
vTest:=True
MESSAGE ("Somebody clicked the mouse")

`Wait Procedure
vTest:=False
ON EVENT CALL ("MouseClicked")
While (Not(vTest))           `Event's waiting loop
.

End while
ON EVENT CALL ("")

```

You would add the command IDLE in the following manner:

```

`Wait Procedure
vTest:=False
ON EVENT CALL (MouseClicked)
While (Not(vTest))
    IDLE                     `Kernel call to sense an event
.

End while
ON EVENT CALL ("")

```

## Macintosh Desktop

Open Document  
Create Document  
Append Document

Document references returned by these functions are of the data type Time.

## Math

**Mod** (value;divider)

The expression "25 modulo 3" can be written in two different ways in 4th DIMENSION:

Variable:=**Mod** (25;3)  
and



Variable:=25%3

The compiler sees a difference between the two: Mod applies to all numerics, while the operator % applies only to Integers and Longints (if the operand of the % operator exceeds the limits of the Longints, the returned result is likely to be wrong). Whenever possible, use %.

## On a List

Subtotal (*data*)

The Subtotal function does not initiate break processing in compiled databases. Use the BREAK LEVEL command to initiate break processing and the ACCUMULATE command to specify what to accumulate for subtotals.

## Strings

Ascii (*character*)

In interpreted mode, you can pass either a non-empty string or an empty string to this function. In compiled mode, you cannot pass an empty string. If you did so, the compiler would not be able to detect an error in compilation if the argument passed to Ascii is a variable.

The Range Checking option displays a message if you use this function with an empty string.

## Structure Access

Field (*field pointer*) or Field (*file number; field number*)  
File (*file pointer*) or File (*file number*)

These two commands return results of different data types, according to the parameters passed to them:

- If you pass a pointer to the File function, the result it returns is a number.
- If you pass a number to the File function, the result it returns is a pointer.

The two functions are not sufficient for the compiler to determine the data type of the result. In such case, to avoid any ambiguity, use a compiler directive.

## Variables and Miscellaneous

Undefined (*variable*)  
 SAVE VARIABLE (*document; variable1; {...; variableN}*)  
 LOAD VARIABLE (*document; variable1; {...; variableN}*)  
 CLEAR VARIABLE (*variable*)  
 Get Pointer (*variable*)  
 EXECUTE (*formula*)  
 TRACE  
 NO TRACE

### Undefined

A variable can never be undefined in compiled mode. In a compiled database, all variables are initialized to nulls before the startup procedure runs. The Undefined function therefore always returns FALSE. If the compiler encounters this function, it generates a warning.

*NOTE: The Undefined function enables you to determine whether you are running in compiled mode. Test any variable for definition immediately after opening the database. If the Undefined function returns TRUE, the database is not compiled; otherwise, it is compiled.*

### SAVE VARIABLE and LOAD VARIABLE

In interpreted mode, you can check that the document exists by testing if one of the variables is undefined after performing a LOAD VARIABLE. This is no longer feasible in compiled databases, since the Undefined function always returns FALSE.

This test can be performed in either interpreted or compiled mode by:

- Initializing the variables that you will receive to a value that is not a legal value for any of the variables
- Comparing one of the received variables to the initialization value after LOAD VARIABLE

The procedure can be written as follows:

```
Var1:="xxxxxx"      `xxxxxx is a value that cannot be returned by LOAD
Var2:="xxxxxx"      `VARIABLE
Var3:="xxxxxx"
Var4:="xxxxxx"
LOAD VARIABLE ("Document";Var1;Var2;Var3;Var4)
If (Var1="xxxxxx")  `Document not found
.
Else                `Document found
.
End if
```

## **CLEAR VARIABLE**

This routine uses two different syntaxes in interpreted mode:

```
CLEAR VARIABLE (variable)
CLEAR VARIABLE ("a")
```

In compiled mode, the first syntax of **CLEAR VARIABLE** reinitializes the variable (set to null for a numeric; empty string for a character string or a text, and so forth), since no variable can be undefined in compiled mode. Consequently, **CLEAR VARIABLE** does not free any memory in compiled mode, except in three cases: Text, Picture, and Array type variables.

For an array, **CLEAR VARIABLE** has the same effect as a new array declaration where size is set to null.

For an array whose elements are Integers, **CLEAR VARIABLE** (Array) has the same effect as one of the following expressions:

```
ARRAY INTEGER (Array;0)    `if it is a 1-dimensional array
ARRAY INTEGER (Array;0;0)  `if it is a 2-dimensional array
```

The second syntax, **CLEAR VARIABLE**("a"), is incompatible with the compiler, since compilers access variables not by name but by address.

## Get Pointer

Get Pointer is a function that returns a pointer to the parameter that you passed to it.

Suppose you want to initialize an array of pointers. Each element in that array points to a given variable. Suppose there are twelve such variables named V1, V2,...V12. You could write:

**ARRAY POINTER** (Arr;12)

Arr{1}:=»V1

Arr{2}:=»V2

Arr{12}:=»V12

You could also write:

**ARRAY POINTER** (Arr;12)

**For** (i;1;12)

Arr{i}:=**Get Pointer** ("V"+**String** (i))

**End for**

At the end of this operation you get an array of pointers where each element points to a variable Vi.

These two sequences are compilable. However, if the variables V1...V12 are not used explicitly elsewhere in the database, the compiler is not able to type them. Therefore they have to be used or declared explicitly elsewhere.

Such explicit declaration may be performed in two ways:

- By declaring V1...V12 through a compiler directive:

**C\_LONGINT** (V1;V2;V3...V12)

- By assigning these variables in a procedure:

V1:=0

V2:=0

V12:=0

## EXECUTE

This command — an historical one in 4th DIMENSION — offers benefits in interpreted mode that are not carried over to compiled mode.

In compiled mode the procedure name passed as a parameter to this command would be merely interpreted. You miss, therefore, some of the advantages provided by the compiler and your parameter's syntax cannot be checked. Moreover, you cannot pass local variables as parameters to it.

EXECUTE can be replaced by a series of statements. Here are two examples:

Assume the following sequence:

```
i:=FctLayout  
EXECUTE ("INPUT LAYOUT(Layout"+String(i)+")")
```

It could be replaced with:

```
i:=FctLayout  
VarLayout:="Layout"+String(i)  
INPUT LAYOUT (VarLayout)
```

Here is another example:

```
$Num:=SelPrinter  
EXECUTE ("Print"+$Num)
```

Here, EXECUTE can be replaced with Case of:

**Case of**

```
:( $Num=1)  
  Print1  
:( $Num=2)  
  Print2  
:( $Num=3)  
  Print3
```

**End case**

The command EXECUTE can always be replaced. Because the procedure to be executed is chosen from the list of the database's global procedures, and there is a finite number of them. Consequently, it is always possible to replace EXECUTE with either Case of or with another command. Furthermore, your code will execute faster.



## TRACE and NO TRACE

These two commands are used in the debugging process. They serve no purpose in a compiled database. However, you can keep them in your procedures; they will simply be ignored by the compiler.

## Pointers with the Following Commands

ADD TO SET  
GOTO RECORD  
GOTO SELECTED RECORD  
APPLY TO SELECTION  
LOAD SET  
SEARCH  
SEARCH SELECTION  
SEARCH BY FORMULA  
DIALOG  
EXPORT TEXT  
EXPORT DIF  
EXPORT SYLK  
CREATE EMPTY SET  
OUTPUT LAYOUT  
INPUT LAYOUT  
GRAPH FILE  
PRINT LAYOUT  
PRINT LABEL  
IMPORT TEXT  
IMPORT DIF  
IMPORT SYLK  
MERGE SELECTION  
CREATE SET  
SORT SELECTION  
PAGE SETUP

These commands have one feature in common: they all accept an optional first File parameter and the second parameter can be a pointer.

In compiled mode, it is easy to retain the optional File parameter. However, the compiler makes an assumption whenever the first parameter passed to one of these commands is a pointer. Since it doesn't know what the pointer is referring to, it assumes that it is a file pointer.

For example, consider the SEARCH command. It has the following syntax:

SEARCH (*/file*;*search argument*;*{\*}*)

where the first element of the search argument must be a field.

## Pointers with the Following Commands

When you write:

**SEARCH** (FieldPtr»=True)

the compiler expects a symbol that stands for a field as the second element in the argument. In fact, it finds the sign "=". It will generate an error message.

On the other hand, if you write:

**SEARCH** (FilePtr»;FieldPtr»=True)

or

**SEARCH** ([File];FieldPtr»=True)

you remove all ambiguity.

If the first parameter to one of these commands is a dereferenced pointer, it must point to a file.



## **7 OPTIMIZATION HINTS**

## OPTIMIZATION HINTS

It is difficult to state once and for all a "good-programming" method. But we wish to stress again the advantages of well structured programs. The capacity for structured programming in 4th DIMENSION can be a great help.

The compilation of a well-structured database can yield much better results than the same effort performed in a poorly-designed one. For instance, if you write a generic procedure to manage  $n$  scripts you will get higher quality results, in interpreted as well as in compiled mode, than in a situation where the  $n$  scripts comprise  $n$  times the same set of statements. In other words, the quality of your programming does have an impact on the quality of the compiled code.

You gradually improve your code under 4th DIMENSION with practice. Frequent use of the 4D Compiler gives you corrective feedback that enables you to reach instinctively for the most efficient solution.

In this chapter, we offer some advice and a few tricks that will save you time in performing simple, recurring tasks.

### Using Compiler Directives to Optimize Code

Compiler directives can help you speed up your code considerably. When typing variables on the basis of their use, the compiler uses the widest data type possible so as not to penalize you. For example, if you don't type the variable defined by the statement: `Var:=5`, the compiler will type it as Real, even if it could be declared an Integer.

Use compiler directives wherever appropriate to type variables as Integer, Longint, or String.

### Numeric Variables

The default data type that the compiler gives numeric variables is Real. But calculations performed on a Real are slower than on an Integer. If you know that a numeric variable will always be an integer, it is to your advantage to declare it through the compiler directives `C_INTEGER` or `C_LONGINT`.

It is good practice, for instance, to declare your loop counters as Integers. Compare the execution times of an empty loop in the following two instances:

```
For ($i;1;50000)  
End for
```

If \$i is not declared by compiler directive (i.e., \$i is Real), the time is 19 seconds. If \$i is declared by compiler directive (C\_INTEGER), it is an instantaneous loop.

*NOTE: These speeds vary according to the model of your Macintosh.*

Some 4th DIMENSION functions return Integers (e.g., Ascii). If you assign the result of one of these functions to an untyped variable of your database, 4D Compiler types it as Real rather than Integer. Remember to declare such variables by compiler directive whenever you are sure that they will not be used in a different context.

Here is a simple example. The function that returns a random value within a given range may be written:

```
$0:= Random % ($2-$1+1)+$1
```

and it will always return an integer.

Written this way, the compiler will type \$0 as Real rather than Integer or Longint. It is preferable, therefore, to include a compiler directive in the procedure.

```
C_LONGINT ($0)  
$0:= Random % ($2-$1+1)+$1
```

The value returned by the procedure will take less space in memory and the procedure will be much faster.

Buttons are a specific case of a Real that can be declared as Longint.

## Strings

The default type assigned to alphanumeric variables is Text. If you write:  
MyString:="Hello"

MyString would be typed as a Text variable by the compiler.

If this variable will be processed frequently, it is worthwhile to declare it using the directive C\_STRING. Processing is much faster with String type variables that have a defined maximum length than with Text variables. Keep in mind the rules governing the behavior of this directive.



*NOTE: If you want to test the value of a character, make the comparison on its Ascii function rather than on the character itself. The regular character comparison procedure considers all of the character's alternatives, such as diacritical marks.*

## Various Observations

This section contains comments on two-dimensional arrays, fields, and pointers.

### Two-dimensional Arrays

The processing of two-dimensional arrays is better managed if the second dimension is larger than the first.

For instance, an array declared as:

**ARRAY INTEGER** (Array;5;1000)

will be better managed than an array declared as:

**ARRAY INTEGER** (Array;1000;5).

### Fields

Whenever you need to perform several calculations on a field, you can improve performance by storing the value of that field in a variable and performing your calculations on the variable rather than the field.

Consider the following procedure:

#### Case of

```

:([Contacts]City="Saratoga")
  Ship:="Blue"
:([Contacts]City="Reno")
  Ship:="Red"
:([Contacts]City="Boston")
  Ship:="FedEx"

```

#### End case

This procedure will take longer to execute than if it were written:

\$Dest:=[Contacts]City

#### Case of

```

:($Dest="Saratoga")
  Ship:="Blue"
:($Dest="Reno")
  Ship:="Red"
:($Dest="Boston")
  Ship:="FedEx"

```

#### End case

If code such as this is placed in a loop and executed repeatedly, the difference in performance could be substantial.

## Pointers

As is the case with fields, it is faster to work with variables than on dereferenced pointers. Whenever you need to perform several calculations on a variable referenced by a pointer, you can save time by storing the dereferenced pointer in a variable.

For instance, suppose you use a pointer to refer to a field or to a variable, MyPtr. Then you want to perform a set of tests on the value referenced by MyPtr. You could write:

### Case of

```
:(MyPtr)=1)
`Sequence1
:(MyPtr)=2)
`Sequence2
```

### End case

The set of tests is performed faster if it were written:

```
$Temp:=MyPtr»
```

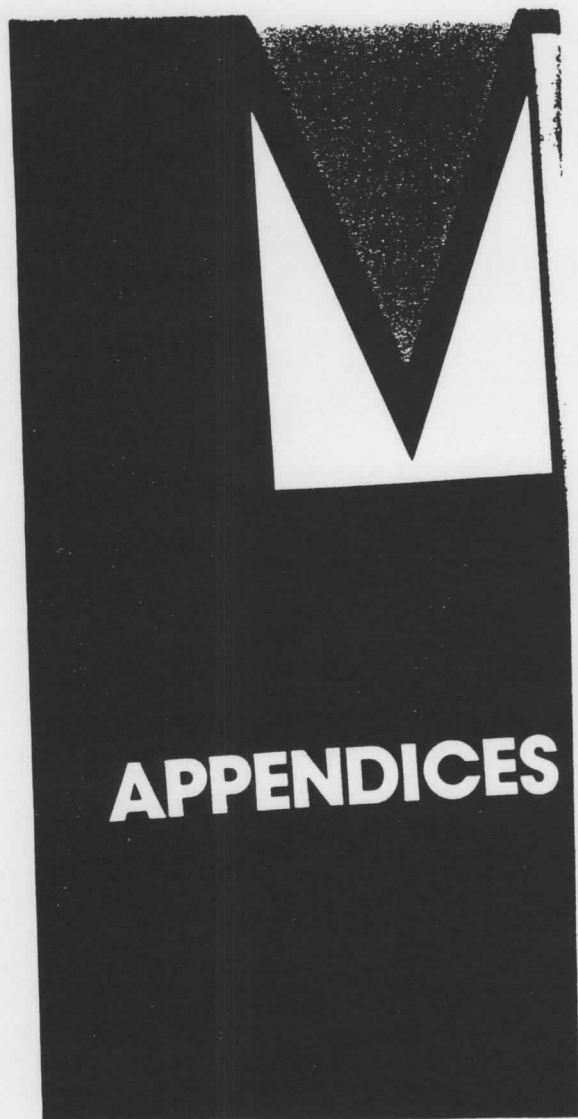
### Case of

```
:( $Temp=1)
`Sequence1
:( $Temp=2)
`Sequence2
```

### End case

## Using Comments

Some of the tricks and shortcuts we suggest may render your code less intelligible to others, and even to yourself later on. Consequently, use comments liberally. Comments are ignored by the compiler and do not appear in the compiled code. While comments tend to slow down an interpreted database, they have no effect on the execution time of a compiled database.



## 4D COMPILER MESSAGES

This appendix lists the messages that can be generated by 4D Compiler. The following types of messages are described:

- Warnings
- Advanced Warnings
- Errors
- Range checking messages
- Compiler messages

Warnings, advanced warnings, and errors are listed in the error file and are displayed in 4th DIMENSION during interactive debugging. Range checking messages are displayed in an alert dialog box while the compiled database is running. Compiler messages refer to the performance of the compiler and are displayed in an Alert dialog box while the compiler is running.

*NOTE: The list of these messages may not be complete because it was written while 4D Compiler was under development. Consult the Read Me file on your program disk for any supplemental information.*

### Warnings

These messages are generated during the typing phase of the compiler. In this section, each message is accompanied with an example of the problem.

*Pointer in COPY ARRAY*

**COPY ARRAY** (Pointer»; Array)

*Pointer in SELECTION TO ARRAY*

**SELECTION TO ARRAY** (Pointer»; MyArray)

**SELECTION TO ARRAY** ([MyFile] MyField; Pointer»)

*Pointer in ARRAY TO SELECTION*

**ARRAY TO SELECTION** (Pointer»; [MyFile] MyField)

*Pointer in LIST TO ARRAY*

**LIST TO ARRAY** ("List"; Pointer»)

*Pointer in ARRAY TO LIST*

**ARRAY TO LIST** (Pointer», "List")

*Using pointer in declaring an array*

**ARRAY REAL** (Pointer»;5)

The command **ARRAY REAL** (Array; Pointer») does not generate this warning. The value of the dimension of an array doesn't have any influence on its own type. In this example, the array referred to by the pointer must have been defined elsewhere.

*Use of Undefined is not advised*

If (**Undefined** (Variable))

The **Undefined** function always returns **FALSE** in a compiled database.

## Advanced Warnings

These messages are generated only if you select the Advanced Warnings option in the Options window. If requested, these warnings appear in the error file.

*The pointer used in this command must refer to an Alphanumeric value*

Pointer»≤2≥:="a"

*The pointer used in this command must refer to an Integer, a Long Integer, or a Real value.*

MyString≤Pointer»≥:="a"

*An array index must be an Integer, Long Integer, or Real Number*

**ALERT** (MyArray{Pointer»})

where the pointer refers to an element in a text or string array.



## Error Messages

These messages are generated in the typing phase and are written to the error file. Each message is accompanied by an example.

These messages are grouped by topic. The topics are:

- Typing
- Syntax
- Parameters
- Operators
- External Procedures
- General Errors

### Typing

*The assignment creates a type conflict*

<b>MyReal:=12.3</b>	<b>`MyReal is Real</b>
<b>MyBoolean:=TRUE</b>	<b>`MyBoolean is Boolean</b>
<b>MyReal:=MyBoolean</b>	<b>`Cannot assign a Boolean to a Real variable</b>

*Changing the length of a string*

**C\_STRING (3;MyString)**  
**C\_STRING (5;MyString)**

*Changing the number of dimensions of an array*

**ARRAY TEXT (MyArray;5;5)**  
**ARRAY TEXT (MyArray;5)**

*Declaring an array without indexes*

**ARRAY INTEGER (MyArray)**

*Expecting a global variable*

**COPY ARRAY (MyArray;")**

*Expecting a numeric constant*

**C\_STRING (Variable;MyString)**

*The type of the variable is unknown  
This variable is used in the procedure:*

The type of variable can't be determined. A compiler directive may be necessary.

*Expecting a text constant*

MyVar := "The weather is nice"

*The following procedure is unknown:*

The line contains a call to a procedure that doesn't exist or no longer exists.

*The length of the character string must be between 1 and 255*

C\_STRING (325;MyString)

*The variable Variable is not a procedure*

Variable (1)

*The variable Variable is not an array*

Variable{5} := 12

*The result of the function is incompatible with the expression*

Text := "Number" + Num (i)

*The data types of the variables in the expression are incompatible*

Integer := MyDate\*Text      `Cannot multiply a date by a text variable

*The index of \${} is not numeric*

\$i := "3"      ` \$i is type Text  
\${\$i} := 5

*The index of an array is not numeric*

IntArray {"3"} := 4      `The index is type Text

*Retrying of the variable Variable of type Text to a numeric type*

Variable := Num (Variable)

*Retyping an array from Integer array to a Text array*

**ARRAY TEXT** (IntArray;12)

if IntArray was declared elsewhere as an Integer array.

*Only pointers can be followed by the sign "»"*

Variable» := 5

if Variable is not of the type Pointer.

*An array is always a global array*

**ARRAY INTEGER** (\$Array;5)

The name of an array always starts with a letter of the alphabet. It cannot begin with the \$ character.

*Using the Text variable Variable as a numeric variable*

Variable := 3.5

*Using a field of the incorrect type*

Variable := [MyFile]MyDate

where [MyFile]MyDate is a Date field and Variable is numeric

## Syntax

*The function does not return a pointer*

Variable := **Num** ("The weather is nice")»

It is not possible to dereference this function

*Syntax error*

**If** (Boolean)

**End For**

`Should be End If

*A closing curly bracket } is missing*

The line contains more opening brackets than closing brackets.

*An opening curly bracket { is missing*

The line contains more closing brackets than opening brackets.

*A closing parenthesis ) is missing*

The line contains more opening parentheses than closing parentheses.

*An opening parenthesis ( is missing*

The line contains more closing parentheses than opening parentheses.

*Expecting a field*

**If (Modified (Variable))**

The Modified function expects a field.

*Expecting an opening bracket {*

**C\_INTEGER (Array 2)**

Array(2):=1

*Expecting a variable*

**C\_INTEGER ([MyFile]MyField)**

*Expecting a numeric constant*

**C\_INTEGER (\${"3"})**

*Expecting a semicolon*

**COPY ARRAY (Array1 Array2)**

*Expecting the closing character reference operator ≥*

MyString≤3 := "a"

*Expecting the opening character reference operator ≤*

MyString≥3 := "a"

*The expression following If must be Boolean*

**If (MyReal)**

if MyReal is a numeric variable.

*The expression is too complex*

Divide your statement into several shorter statements.

*Procedure is too complex*

The procedure contains more than 600 different cases or more than 100 If...End if structures.

*Reference to an unknown field*

Your procedure, possibly copied from another database, contains an expression that refers to a non-existent field.

*Reference to an unknown file*

Your procedure, possibly copied from another database, contains an expression that refers to a non-existent file.

*A pointer cannot be defined in this expression*

Pointer := »Variable + 3

*Incorrect use of the character reference operator*

MyReal≤3≥ or MyString≤Variable≥

where MyReal is a numeric variable and Variable is not a numeric variable.

**Parameters***This result of the function cannot be a parameter of this procedure*

MyProcedure (Num(MyString))

if MyProcedure expects a Boolean expression.

*This value cannot be a parameter of this procedure.*

MyProcedure (3+2)

if MyProcedure expects a Boolean expression.

*Type conflict regarding \$0*

C\_INTEGER (\$0)

\$0 := FALSE

`\$0 is Integer

`\$0 is now Boolean

*Type conflict regarding the generic parameter \${}*

C\_INTEGER (\${3})

For (\$i;3;5)

  \${\$i} := String (\$i)

End For

`\${\$i} is now Text



*The command does not accept a parameter*

**Current Date** (MyDate)

*The command requires at least one parameter*

**DEFAULT FILE**

*The variable MyString cannot be a parameter of this procedure*

**MyProcedure** (MyString)

if **MyProcedure** is expecting a Boolean parameter.

*The type of the parameter \$1 in the called procedure is different than the type of the parameter in the calling statement*

**Calculate** ("3+2")

The parameter is type Text

with the directive **C\_INTEGER(\$1)** in **Calculate**.

*The data type of the parameter passed doesn't match the type of parameter expected*

**Print** ("LaserWriter")

if in the procedure **Print**, \$1 is numeric.

*Retyping a numeric parameter \$1 to text type*

**\$1 := String** (\$1)

*A parameter cannot be an array*

**MyProc** (MyArray)

To pass an array in a procedure, you need to pass a pointer to the array.

*A parameter cannot be used in calling this command*

**RECEIVE VARIABLE** (\$1)

## **Operators**

*Addition cannot be performed on Boolean fields.*

**Boolean2 := Boolean1 + TRUE**

*Not expecting the > operator*

**SEARCH** ([MyFile];[MyFile]MyField=0;>)

*The two arguments are not comparable*

If (Number = Picture2)

where Number is a LongInt and Picture2 is a Picture.

*Negation cannot be used in this expression*

Boolean := -FALSE

## External Procedures

*The external procedure MyExternal is poorly defined*

The definition of the external routine is incorrect.

*The number of parameters passed to the external routine is too large*

## General Errors

*Two procedures have the same name: Name*

To compile your database, all of the global procedures must have different names.

*Internal error No. xx*

If this message appears, call ACIUS or ACI Technical Support and report the error number.

*Cannot determine the type of Variable.*

*This variable is used in the P1 Procedure.*

The Variable type cannot be determined. A compiler directive is necessary.

*The database contains more than 32K of global variables*

You must reduce the number of global variables. Consider using local variables. This message gives the amount of memory used by your global variables, except for arrays and fixed strings (declared by the C\_STRING directive).

*REMINDER: Picture and text variables account for only 4 bytes each.*

*The database contains more than 32K of local variables*

You must reduce the number of **local variables** in the procedure. This message gives the amount of memory used by your local variables, except for fixed strings (declared by the **C\_STRING** directive).

*The procedure is larger than 32K*

Divide your procedure into **smaller procedures**. Chances are you are not writing tight code.

*The original procedure is damaged*

The procedure is damaged in the **original structure**. Delete it or replace it.

*4th DIMENSION command is unknown*

The procedure is damaged or the **version of the compiler** you are using was released before the command was added to 4th DIMENSION.

*Retyping the variable Variable in the format Format*

The message appears if you give, for example, the name OK to a variable of the type Graph in a layout.

*A function and a variable have the same name: Name*

Rename either the function or the **global variable**.

*An active object has the same name as a function: Name*

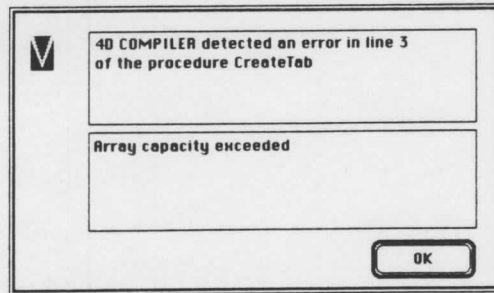
Rename either the active object or the **function**.

*An external procedure and a variable have the same name: Name*

Rename either the external procedure or the **variable**.

## Range Checking Messages

These messages are displayed in 4th DIMENSION while the compiled database is running. These messages appear in the following window:



*Array capacity exceeded*

```
MyArray{17} := 2.3
```

MyArray is an array with 5 elements at the time the above statement is executed. This message will appear if you try to access the {17} element in the array.

*The parameter cannot be passed.*

Using the \$4 local variable when only three parameters have been passed to the current procedure.

*The pointer is not correctly initialized.*

```
MyPointer» := 5
```

if MyPointer hasn't yet been initialized.

*String in which the maximum length is too short.*

```
C_STRING (5;MyString1)
```

```
C_STRING (10;MyString2)
```

```
MyString2 := "TheString"
```

```
MyString1 := MyString2
```

String2 has 9 characters, but String1 can store only 5

*The string index isn't valid (too large or negative).*

```
i:=30
```

```
MyString[i] := MyString2
```

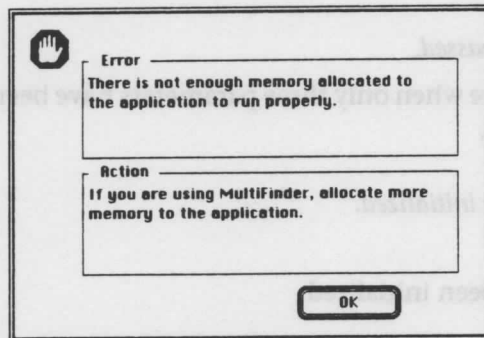
*The passed string in the parameter is empty or not initialized.*

```
MyString := ""
```

```
Number := Ascii (MyString)
```

## Compiler Messages

When its work environment is not optimal, 4D Compiler displays messages that encourage you to improve its environment. These messages are displayed in the alert box shown below.



The Error area describes the problem. The Action area describes a way to fix it.

## CUSTOMIZING APPLICATION ICONS

Using 4D Compiler, you can merge your database with a 4D Runtime to create a stand-alone (double-clickable) application. You can do this simply by selecting the option "Merge with Runtime" in the Options window. For complete information about this option, see Chapter 2.

When you merge your database with a runtime, 4D Compiler creates a stand-alone application with a generic application icon. You can customize this icon by following the procedure described in this appendix. You can also customize the icons for all the other documents that are created by a 4th DIMENSION database, i.e., the data file, quick reports, labels, saved searches, and so forth.

This procedure uses:

- A resource editor, such as ResEdit™
- The Resources database, located on your 4D Compiler program disk
- The uncompiled version of your database

*NOTE: ResEdit is a resource editing utility available from Apple Computer, Inc. This appendix assumes you are familiar with ResEdit.*

To customize your database icon, follow these steps:

**1. Copy the Resources database to your hard disk and open the structure file using ResEdit.**

**2. Copy the following resources from Resources to the structure file of the database to be compiled.**

- SIG\*
- BNDL
- FREF
- ICN#

Use standard copy and paste operations.

*NOTE: The FREF resource must not be modified.*

In the next series of steps, you will specify the Creator of your application.



**3. Open the BNDL resource and replace the OwnerName with the 4-character Creator for your application.**

The Creator must be different from the Creator of any other Macintosh application on your hard disk. If you inadvertently use the Creator of an existing application, the Finder might use the other application's icon instead of yours.

If you intend to market your application, you need to check with Apple Computer, Inc. to ensure that your Creator is unique, i.e., not used by any other published Macintosh application.

**4. Open the SIG\* resource and enter the 4-character Creator for your application.**

In the next series of steps, you will use ResEdit to customize your application's icon.

**5. Open the ICN# resource.**

**6. Open the icon with the identification number 128.**

ResEdit opens an editing window in which you can edit the icon. The bit-map on the left is the icon; the bit-map on the right is its "mask."

**7. Modify the icon as you like.**

**8. When you finish modifying each resource, save your changes by choosing Save from the File menu and close the window belonging to that resource.**

This step completes the process of customizing your application's icon. You can also customize the other files that are created by your application while it is running — the data file, quick reports, labels, searches, and so forth.

To customize the icons belonging to these files, follow these steps:

**1. Open the appropriate ICN# resource and modify the icon using the bit-map editing tools.**

**2. When you have finished customizing each icon, choose Save from the File menu.**

**3. Close your application and quit from ResEdit.**

You are now ready to compile your database with the Merge with Runtime option.

After the compiler creates your custom application, rebuild the desktop. In this process, the Finder adds your custom icons to its desktop file. After the desktop is rebuilt, your custom icons will appear.



*NOTE: You rebuild your desktop by choosing Restart from the Finder's Special menu and holding down the Option and ⌘ keys. A dialog box will ask you whether you want to rebuild your desktop.*

## Symbols

- % symbol 84
- () (parentheses) 104
- | (curly bracket) 103
- ≤ (character reference operator) 104, 105

## Numerics

## 4D Compiler

- compilation options 22–28
- installing 10
- memory requirements 10
- menus 18–21
- setting cache memory 10
- typing variables 52–53
- using with MultiFinder 17–18

## 4D COMPILER window 13, 28–29

- Abort button 32
- Pause button 32

## 4D Runtime 3, 23

- 68000 3, 27
- 68020/30 3, 27
- 68881/82 3, 27

## A

- ABORT command 82–83
- accelerator cards
  - third party 27
- ACCUMULATE command 84
- ADD TO SET command 89
- Advanced Warning messages 47
- advanced warnings 100
- ALERT command 100
- Append Document function 83
- APPLY TO SELECTION command 89
- array declaration statements 29
- ARRAY INTEGER command 101, 103
- ARRAY POINTER command 87
- ARRAY REAL command 100
- ARRAY STRING command 59
- ARRAY TEXT command 101, 103
- ARRAY TO LIST command 79, 80, 100
- ARRAY TO SELECTION command 79, 80, 99
- array variables
  - conflicts with 66

## array-related commands

- using pointers with 81
- arrays 30, 79–81, 109
  - changing the number of dimensions 67
  - implicit retyping of 68
  - in symbol table 37
  - string 68
  - two-dimensional 95

## Ascii function 84, 94, 109

## B

- BREAK LEVEL command 84
- break processing
  - in compiled databases 84
- buttons 94

## C

- Case of structure 81, 88, 95, 105
- CLEAR VARIABLE command 85, 86
- comments 9, 96
- communications 81–82
- Compilation options 22–28
  - Advanced Warning 26
  - Compiled Database Name 22
  - Error File 24
  - Merge with Runtime 23, 29
  - Processor Type 27
  - Range Checking 25
  - Script Manager 26
  - Symbol Table 25
- compilation pass 31
- compilation process 28–29
  - compilation pass 31
  - copying the database 29
  - directive typing pass 30
  - global typing pass 30
  - interrupting 32
  - local typing pass 30
  - typing variables 29
- compiled mode 6–7, 89
- Compiler directives 29, 30, 52, 54–56, 56–59, 64, 65
  - C\_BOOLEAN 54
  - C\_DATE 54
  - C\_INTEGER 54, 57, 93, 104, 105
  - C\_LONGINT 54, 57, 93
  - C\_PICTURE 54
  - C\_POINTER 54

## C\_REAL 54

- C\_STRING 54, 57, 59, 65, 94, 101, 102, 107, 109
- C\_TEXT 54
- C\_TIME 54
  - optimizing code with 56, 93–96
  - placing 58
  - using with the interpreter 57–58
  - when to use 54–56

## compiler messages 110

## COPY ARRAY command 79, 99, 101, 104

## counters 56, 94

## Create Document function 83

## Current Date function 106

## D

- data entry 82
- DEFAULT FILE command 106
- DIALOG command 89
- double-clickable applications 3
  - creating 23
  - custom icons for 111
  - generic icon for 24, 33

## E

- empty strings 84
- error file 5, 7, 9, 24, 26, 40–45, 101
  - as text 43
  - errors linked to a specific line 41–42
  - general errors 40–41
  - structure of 43
  - using 43–45
  - using interactively 44
  - warnings 42–43, 81
- error messages 101–108
- errors
  - displaying 32
  - exceptions 82–83
- EXECUTE command 85, 88
- EXPORT TEXT command 89
- external procedures 28, 71, 107, 108
  - passing parameters to 71

- FIELD ATTRIBUTES command 56
- Field function 84
- fields 95
- File function 84
- File menu 18-21
  - Close command 21
  - New command 18
  - Open command 19
  - Quit command 21
  - Recompile command 21
  - Revert to Saved command 21
  - Save as command 21
  - Save command 21
- For...End for structure 8

**G**

- general errors 107-108
- generic parameters 74, 105
- Get Pointer function 85, 87
- global procedures
  - in symbol table 39
- global variables
  - in symbol table 37-38
  - typing 29, 30
  - typing conflicts with 63-65
- GOTO RECORD command 89
- GOTO SELECTED RECORD command 89

**I**

- IDLE command 82-83
- If...End if structure 104, 107
- implicit retyping 64
- interactive debugging 3, 7, 9, 21, 44-45
  - naming error file for 24
- interpreted mode 6, 8, 14, 28, 52, 74, 84, 86

**K**

- kernel calls 82

**L**

- layout variables 68-70
  - button 68
  - check box 68
  - dials 68
  - external object 69
  - graph 69

- pop-up menus 69
- radio button 68
- radio picture 68
- ruler 68
- scrollable areas 69
- simple variables 69
- thermometer 68
- LIST TO ARRAY command 79, 80, 99
- LOAD SET command 89
- LOAD VARIABLE command 85
- local variables 56, 58, 72, 95, 96, 106, 109
  - in symbol table 39
  - typing 29, 30
  - typing conflicts with 66
- loop counters 56, 94

**M**

- machine language 6
- Macintosh desktop 83
- math 83
- mathematical coprocessor 3, 27
- Mod function 83
- Modified function 104
- MultiFinder 17, 21
- multi-syntax commands 56

**N**

- NO TRACE command 85, 89
- Num function 103, 105
- numeric variables 4

**O**

- On a list 84
- ON EVENT CALL command 82-83
- ON SERIAL PORT CALL command 82-83
- Open Document function 83
- operators 106-107
- Options window 12, 19, 20, 22, 111

**P**

- parameter passing 72
- parameters 105-106
- passwords 28

- pointers 55-56, 70, 81, 82, 84, 87, 89-90, 96, 99-100, 103, 105, 109
  - using to avoid retyping 72-74
- poorly-written procedures 4
- proc.ext file 71
- project 12, 17, 19

**Q**

- quick report variables 76

**R**

- Range Checking 5, 25, 45-47, 70, 84
- RECEIVE VARIABLE command 81, 106
- Recompiling
  - using a project 20-21
- reserved variables 75
- Resources database 111

**S**

- SANE environment 56
- SAVE VARIABLE command 85
- SEARCH BY FORMULA command 89
- SEARCH command 89-90, 106
- SEARCH SELECTION command 89
- SELECTION TO ARRAY command 79, 80, 99
- SEND VARIABLE command 81
- Startup procedures 8, 58
- String function 105, 106
- string variables 65
- strings 4, 84
- structure access 84
- structure file
  - placing externals in 71
- structured programming 93
- Subtotal function 84
- symbol table 5, 37-39
  - definition of 52
  - list of global variables 37
  - list of local variables 39
- syntax errors 103-105
- System file
  - placing externals in 71
- system variables 75

## T

TRACE command 85, 89

Type function 82

typing

numeric variables 53, 93–94

strings 94–95

text variables 56

typing phase 29–31

conclusion of 31

## U

Undefined function 85, 100

Use menu

Next Compiler Error command

7, 44–45

Stop Browsing Error File com-

mand 44–45

## V

variables and miscellaneous 85

## W

warnings 42–43, 99–100

displaying 32